C言語を256倍使うための本

Microsoft C Compiler

福崎俊博・梅原 系・山田伸一郎 共著

アスキー出版局

C言語を256倍使うための本

Microsoft C Compiler

福崎俊博・梅原 系・山田伸一郎 共著

アスキー出版局

■商標

＊MS-DOS，Quick C は，米国 Microsoft 社の登録商標です．

＊Microsoft C Compiler は米国 Microsoft 社の商標です．

＊Turbo C は米国 Borland International 社の登録商標です．

＊UNIX オペレーティングシステムは AT&T のベル研究所が開発し，AT&T がライセンスしています．

＊その他，本書で使用する CPU 名，システム名，製品名等は一般に各開発会社の商標です．なお，本文中では TM，®マークは明記していません．

本文イラスト　CHIPS

はじめに

C言語に関する書籍はこれまでにも数多く出版されてきた．その出版点数の多さといったら，今やダイエットや占いの本と肩を並べると言っても過言ではないくらいなのだ(過言かもしれない)．しかし，しかしである．そんな中で，中級以上のCプログラマがさらにステップアップしようというとき，参考になる書籍は一体どれほどあるだろうか．その点，欲求不満を感じている方もきっと多いはずだ．

本書は，まさにそうした方々のために書かれている．C言語と一口に言っても広く深い世界なので，通常の解説書だと原則的な事柄はカバーできても，意表をついた細かいテクニックや，初級者では理解しにくい高度な技まではなかなか手が回らないのが実情なのだ．本書には，そうしたテクニックやノウハウをぎっしりと詰め込んである．本書は一応MS-Cをターゲットにしているが，必ずしもMS-Cの場合に限らない題材が多いので，他の処理系をお使いの方にも安心して読んでいただけると思う．また，本書は「256倍シリーズ」の一冊なので，もちろん単なるテクニック集には終わらせず，誰も書かなかったCにまつわるいろいろな話題も紹介しておいた．

そもそもこの256倍シリーズは，「真面目な内容だからといって，真面目くさって書かなくてもよいのではないか」というコンセプトから出発したものだった，……ような気がする(今となっては真相は闇の中である)．本書でも，我々はその路線を忠実に守った．定価が安い割には内容は豊富で高度(?)，著者は泣いても読者は笑うという256倍シリーズならではの心意気を感じ取っていただければ幸せである．

さて，本書の執筆分担だが，1章と2章は福崎が，3章は山田，6章は梅原がそれぞれ執筆し，4章と5章は主に山田と福崎が分担して執筆した．

本書の執筆に当たっては，日経MIXでc.会議の議長を務める水居浄氏にお世話になった．また，池田けんしろう，上野和明，榊正憲，柳楽直樹の各氏にはネタ出しや技術的な面で協力していただいた．編集担当の中島氏には各種調整や食事の面でお世話になった．深く感謝しておきたい．

1990年盛夏

著　者

本書の構成

本書は，大きく分けて3つの部分から構成されている．まず，最初の2つの章はC言語に関する総論で，続く3つの章はCのプリプロセッサ，コーディング，ライブラリといういわば各論である．そして，最後の6章は，総仕上げとして実際のプログラム開発を例にとった解説に当てている．

各章の具体的な内容は次のとおりである．

Cウィザードへの長い道のり

まず本書の手始めとしてCを使うための必要条件や気構えを述べる．次いでCの歴史や，MS-Cをはじめとする処理系の紹介をするとともに，ANSI標準について解説している．

C言語の常識と非常識

移植性の神話(「Cで書いたプログラムには移植性がある」という嘘八百)を槍玉に上げた後，プログラミングスタイルや変数のネーミング法，プログラムの音読法など，悪名高い(?)Cプログラマのこだわりアラカルトを紹介する．

プリプロセッサ道場

普段あまり日の当たらないCのプリプロセッサを取り上げる．マクロ定義に関する落し穴を詳しく解説し，プリプロセッサによるベースドポインタの実現方法などの大技から便利な小技まで，各種マクロを紹介する．

標準コーディングテクニック

Cでコーディングする場合に注意すべき点，ポインタや構造体の徹底使いこなし，アイディアを活かした技など，実際のプログラミングに役立つ話題を豊富に解説している．また，ポインタはintではないという常識と，それに果敢に立ち向かうひとりの青年の主張も紹介する．

ライブラリ活用のヒント

数あるライブラリ関数のうち，主として入出力とメモリ管理を取り上げて解説する．printf()やscanf()の問題点，またメモリ管理に関連して，farポインタを使う上での留意点などについても言及する．

もっと速く，もっと小さく

本書の総まとめとして，高速に動作し，サイズも小さい高品質なプログラムをどのように開発するかを解説する．そこでcatを例にとり，数種類のバージョンのcatを実際に開発している．遅くサイズも大きかった最初のバージョンが，徐々に優れたプログラムに仕上がっていく様子がよくわかるはずだ．

目次

C ウィザードへの長い道のり

C 言語は難しい 11
C 言語物語 14
UNIX ワールドの暴れん坊—gcc— 17
安い，速い，優秀？—MS-DOS 用の処理系— 18
高いが安心，一流ブランド—MS-C— 20
ANSI C なんて大嫌い？ 24

C 言語の常識と非常識

C 言語には移植性がない 31
スタイルからキメてみる 34
C を使う覚悟 52

プリプロセッサ道場

マクロ定義の光と影 55
マクロ・マクロ・マクロ！ 65

標準コーディングテクニック

switch 文をきちんと使う 87
関数まわりのコーディング 91
データ型に潜む罠 103
究極の添字 109
ポインタの完全制覇 116
構造体を 256 倍使う 126

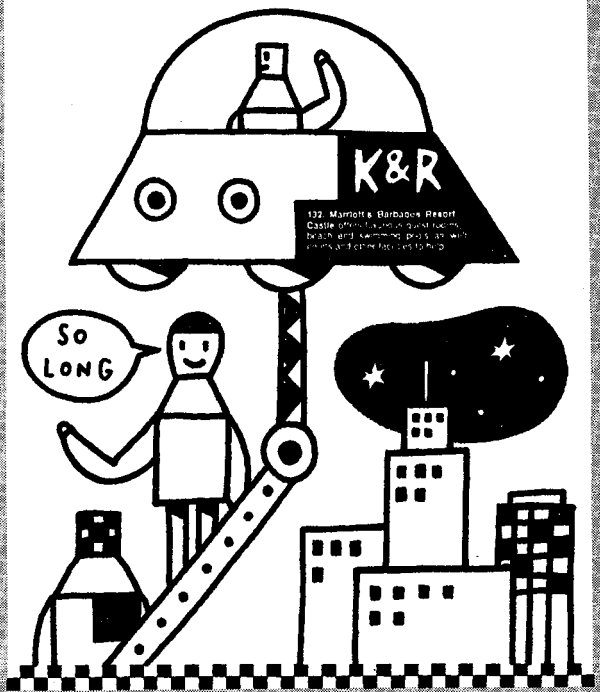
ライブラリ活用のヒント

入出力関数 139
メモリモデルとメモリ管理 163
ほんとに使うライブラリ関数はこれだけだ! 174

もっと速く，もっと小さく

あるTOOLSプログラマのこだわり 181
高速化との果てしない戦い 185
正しい仕様をめぐる闘い 198
やっぱり気になるスリーサイズ 206
キャウ・cat・キャウン 210
仕様を煮詰める 235
ウィザードへの道はまだまだ長いぞ 244

Cウィザードへの長い道のり



C言語が全盛だといってCを使い，そのままやたらと年月を重ねたところで，誰もがCウィザードになれるわけではない．ウィザードへの道のりは，長く遠く険しいのだ．この章では安易なC言語ブームを戒め，Cプログラマの心構えを説くとともに，各種のC処理系やANSI標準について簡単に紹介する．

C言語は難しい

C言語がブームである.今さら何言ってんのと思われるだろうが,このブームはもう数年続いており,単なる流行では片付けられない状況になっている.本屋に行けば解説書が山のように置いてあるし,「日本におけるC言語の学習者人口は英語に次いで多い」というのもあながちウソではないように思える(でも多分ウソ).ケント・ギルバート外語学院の向こうを張ってデニス・リッチーC言語学院でも開校すれば,大繁盛間違いなしというほどのブームなのだ.

しかし敢えて言ってしまおう,

初心者はC言語を使ってはいけない!

ヴィトンやグッチじゃあるまいし,「みんなが使ってるから」とか「流行だから」とかいう理由で安易にC言語というブランドにすがっても,良質なプログラムがバカスカ書けるようになるわけがない.南青山で編集の仕事をしてるからといって,女にもてると思ったら大間違いなのと同じなのだ*1.

*1編集子注:私のことではありません(きっと).

C言語以前にマスターすべきこと

いずれC言語を使ってみたいプログラミング初心者なら，まずPascalとアセンブラを勉強すべきだ．C言語ではいい加減に書ける部分でもPascalならきちんとエラーにしてくれるし，プログラムの構造をプログラム上で明確化してドキュメント性に気を使うクセが身に付くはずだ(Pascalはタイプ量が多いのでキーボードの練習にもなる)．またアセンブラに親しんでおけばコンピュータについていろいろ勉強になるし，メモリ中での値の格納や参照のされ方など，Cプログラミングに絶対欠かせない知識が得られるだろう．Cをマスターするのに別の言語を学ぶというのは遠回りのようだが，いきなりCで構造不明のスパゲッティプログラムを書きまくってデバッグ不能になったり，いつまで経ってもポインタがうまく使えないでいるよりは，はるかに近道だと思う．もちろん，カーニハン，プローガー共著の『ソフトウェア作法』(共立出版刊)などの参考書を読んで，プログラムの作り方を勉強するのも忘れてはならない．

また，C言語をマスターする上では，

Cは自らハックするものを助く

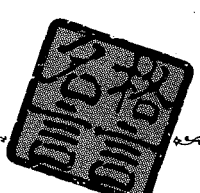
という言葉を肝に銘じておきたい．優秀なCプログラマでありながら某大学の大学院に通う山田君(仮名)は，調べりゃすぐにわかりそうなことを学部生がいちいち質問してくると言って嘆いている．それなら「マニュアルを読め」とか「自分で調べろ」と一喝すれば済むような話だが，山田君(仮名)は人がいいので，懇切丁寧に教えてしまう．ひょっとしたら，山田君(仮名)はそんな自分に腹を立てているのかも知れない．

それはともかく，自分でろくに努力をしないでマスターできるほどC言語は簡単ではない．マニュアルやK&Rを何度も読み，コンパイラが吐いたコードをデバッガを駆使して解析し，というような努力を重ね，何事も自

分の力で解決しようという根性が必要なのだ．夕食に何を食べるかで頭がいっぱいのＣウィザードに，

ねえねえ，
CTRL-Cってどうやってトラップすればいいの?

などとふざけた質問をしてはならないのである．



猫に小判

＜解説＞

代表的なＣプログラムのcatには，たくさんのオプションがある．いずれも，プログラマが便利だと思って追加したオプションである．しかし，それらのオプションのほとんどは使われない．つまり，プログラマが便利だと考えて付けた機能もユーザーにとっては用の無いものがほとんどだということ．

C言語物語

C言語とUNIXオペレーティングシステムが密接な関係にあるのは周知のとおりだろう．といっても，UNIXが最初からCで記述されていたわけではない．初期のUNIX(1971～2年頃)は，DEC社製ミニコンピュータPDP-11(一番初めはPDP-7)用のアセンブラで書かれていた．しかし，UNIXの開発者であるKen ThompsonやDennis Ritchieは，かなり早い時期からシステムを高級言語で記述しようとしていたようで，オペレーティングシステム自体の開発と並行してC言語の研究も進めていた．そして1973年にCに一通りの機能が揃った時点で，UNIXのカーネルやシェルがCで書き直されたのである(UNIX Version 4)．

UNIXのVersion 6あたりまでに付いてきたCコンパイラの言語仕様は，現在のCの言語仕様とは若干異なっている．有名な相違点として，たとえば「xから3を減じる」式は，今なら，

```
x -= 3;
```

と書くところが，

```
x =- 3;
```

のように書けた．この書式だと，xから3を減じるのか，xに−3を代入するのか曖昧な場合があるため，現在の書式に変更されている．また，変数の初期化にしても，今なら，

```
int x = 1;
```

のように書くが，以前は，

```
int x 1;
```

だった．しかし，〝int x (1 +2) ;〟のような初期化をすると，〝int x() ;〟のような関数宣言とまぎらわしく，コンパイラが困るためにこの書式はボツになったという(『K&R 第1版』参照)．

また，ライブラリ関係では，Thompson や Ritchie と同じベル研の Mike Lesk という人(lex の開発者)が Portable C Library を作成している．これが Ritchie によって書き直され，標準入出力ライブラリとして有名になった．

さて，UNIX の C コンパイラは，Version 7(V7)の UNIX になって K&R の仕様が完全に満たされ，現在の各種 C 処理系の源流となった．今なら C コンパイラの宣伝文句としては「ANSI 準拠」というのがポピュラーだが，少し前までは「K&R 準拠」とか「V7 C 準拠」と広告に書いておけば，実際に使ってウソだとわかるまでは一応みんな安心したものである．

それでも数年前までは，Whitesmith C や BDS C，Small C など独自の処理系も存在していた．こうした処理系はライブラリが V7 の仕様と全然違ったりして不便だったが，Whitesmith C などは開発された時期が古かったということもあって，AT&T の著作権に気を使ってわざと非互換にしたというのが真相らしい．しかし，このような独自の処理系もいつのまにかなくなっちゃったり，UNIX 互換になったりして，プログラマとしては，

めでたしめでたし

なのである．

ところがMS-DOSでCを使う場合には，言語やライブラリの仕様がK&R互換だろうがV7互換だろうが，セグメントに気を使わなければならないとか，リニアアドレスが狭くて巨大な配列が素直に作れないとか，変な制限がたくさんある．アジアの島国と同様，狭い世界には独特の文化が生まれるもので，MS-Cを例にとって言えば，標準的なCにはないfarやnearというキーワード，各種のメモリモデル，メモリモデルごとに用意されたライブラリなどが存在しており，狭いながらもそれなりにゴージャスな気分に浸れる，わけはなく，

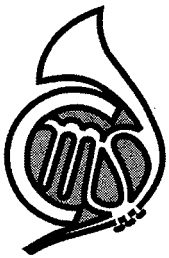
家は買えないからせめて車だけでも

みたいな感じで少し悲しくなってしまう(愚痴を言っても仕方ないけど)．

親の心子知らず

＜解説＞

標準入出力を使わない，あるいは潰してしまうこと．せっかく親プロセスがリダイレクトしたストリームに対して失礼である．親プロセスがストリームを使ってプロセス間通信をしようとしている場合には，親プロセスの期待を裏切っていることにもなる．ひどいプログラムになると，標準入出力をクローズしてしまったりする．嘆かわしい．



UNIXワールドの暴れん坊—gcc—

UNIXの動作するワークステーションでは，最近gccというCコンパイラが非常に人気を集めているようだ．gccは，GNUというソフトウェア体系に含まれるCコンパイラである．GNUにはUNIX互換のツール類が用意されており(GNUは〝GNU's Not UNIX〟の略)，MITのAIラボを飛び出したRichard Stallmanの主宰する，Free Software Foundationという団体が無料で配布している(ただしPDSではない)．

gccが人気を集めているのは，無料で手に入るということもあるかもしれないが，ANSI標準をサポートしつつも独自の拡張が施されているし，出力コードの質も高いからだろう．ただし，かなりの頻度でバージョンアップされるので，各サイトのPDS担当は死んでいるという話もある．

ただし，gccを動作させるにはかなりのメモリ(条件によって違うが，1.5～2Mバイト以上)が必要なので，残念ながらMS-DOSでは実行できない．国産のパーソナルコンピュータでも，CPUに68000を使ったX68000用にはgccが移植されているようだが，MS-DOSを使っている我々としては，将来MS-DOSで普通に数Mバイトのメモリが使えるようになるのを待つか，ワークステーションを買うために貯金を始めるか，思い切って(?)X68000を買ってしまうほかに手はないようだ．

Richard Stallmanたちの Free Software Foundationは，自ら優れたソフトウェアを無料で配布しつつ，ソフトウェアの共有を目指して商業化に反対している．その思想については，「GNU Manifesto」という短い文章にまとめられているので，ぜひ一読をお勧めしたい(共立出版刊『GNU Emacsマニュアル』にも邦訳が掲載されている)．

安い，速い，優秀？ —MS-DOS用の処理系—

さて，パーソナルコンピュータ用のC処理系の種類は多いが，MS-DOS用としては，日本ではMS-C，QuickC，Turbo Cの3つに人気が集中しているようだ．

QuickCは位置付けとしては「C言語ないしMS-Cの入門用」とされているが，実際にはライブラリも本家MS-C相当のものが付いてくるし，MS-C ver.5.10で使えるメモリモデルもすべてサポートされている．おまけにMS-C ver.5.10では利用できない統合環境やインラインアセンブリ機能，PC-9801シリーズ用のBIOSライブラリ，グラフィックスライブラリまで付いていて，コンパイル速度が速く，しかもMS-Cより圧倒的に安価なのである．これでは，

MS-C ver.5.10を買った人の立場はど〜なる

みたいな気がするが，MS-C ver.5.10ほど優れたコードが出力されるわけではないので，QuickCで快適に開発して，最終的にはMS-Cでコンパイルして完成させるというやり方が(お金があれば)ベストだろう．後述するように，ver.6.00になってMS-Cにもインラインアセンブラの機能などが追加されている．

一方，Turbo Cに人気があるというのもうなずける．なにしろ価格が安い(QuickCが発売されてからはそれほど割安感がなくなったとはいえ)ので，アマチュアのプログラマにも手軽に購入できるからだ．かつて一世を風靡したTurbo Pascal以来のTurboファンも多い．そして，コンパイルがMS-Cなんかに比べてめちゃくちゃ速いというのも定評のあるところ

である．ただ，ライブラリや言語仕様の一部で互換性が問題になったり(特殊な拡張がされている)，Microsoft が Turbo C を出しているわけではない悲しさか，OS/2 などの新しい環境への対応の遅さを不安として指摘する向きもあるようだ．

このほか，価格はやや高めだが，LSI C(8080 用で，CP/M 時代にはその出力コードは「芸術的」とすら言われた)の流れを汲む処理系，LSI C-86 を愛用している人も多い．また，IBM-PC 用としては，各種のベンチマークで非常に高い成績を収め，最近かなりの注目を集めている Watcom C 7.00, ANSI 標準の承認よりもリリースが早いのに，なぜか ANSI 標準対応度 100 %を誇る Lattice C 6.0(MS-C ver.6.00 ですら，ANSI 対応度は 100 %までいっていない)など，日本ではそれほど馴染みがないが，面白そうな処理系がかなりあるようだ．

言語同段(げんごどうだん)

＜解説＞

C や Pascal はインデント（段下げ）が一般的である．しかし FORTRAN などの古いプログラミング言語には「7 カラム目の美学」があり，きっちりと左端を揃えたプログラムが美しいとされている．このプログラミングスタイルを C プログラムでも守り続けようという人々や，インデントの概念を理解できない人々の書いたプログラムは読みにくくてしようがない．このプログラムを言語同段なプログラムという．転じて，許し難い行為のこともいう．

高いが安心，一流ブランド
—MS-C—

さて，MS-Cである．MS-Cが日本で使われ始めてもう何年にもなるが，最初に英文の暫定マニュアルを読んだときの胸騒ぎは今でも忘れられない．当時，MS-DOSで使える処理系はすでに国内で何種類か販売されていたものの，ライブラリがとてつもなく貧弱だったり，コンパイラに許せないバグがあったり，言語仕様自体がかなりのサブセットだったりして，けっこう苦労したものだ(それでもLatticeはまずまず使えた)．しかも，親切に(?)バグレポートを送ってやったら，電話の向こうでさんざん怒った挙げ句，

ソフトウェアにはバグが付きものである

などと居直ったところもあったくらいなのだ．

そんな中で出会ったのがMS-Cの暫定マニュアルだった．パラパラとページをめくってみると，まるで砂漠で泉を見つけたような気分になった．言語仕様もライブラリの仕様も，これまでのMS-DOS用の処理系と比べて抜群に充実していたのである．もっとも，借りてきた処理系のβ版はコンパイルの実行途中でなぜかいつもお亡くなりになってしまって，実際にはテストできず，当時随分口惜しい思いをしたものだ．製品が実際に日本で発売されたのは，それから1年以上も経ってからだった．

プロがMS-Cを選ぶわけ

そんなわけで，現在，MS-Cを使っている人はかなり多い．特に，MS-Cの場合には，プロの割合が大きいという特徴がある．その理由は，大きく分けて2つあると思う．

- いろいろな意味でMS-Cなら安心
- MS-Cは価格が高いので，プロならともかくアマチュアはみんなTurbo CやQuick Cを買う

MS-Cを使うことで得られるさまざまな安心感というのは，実際に多くの人が感じているだろう．まず1つには，当り前の話だが，MS-Cの開発元がMicrosoftだということだ．MS-DOSを普及させ，Windowsを開発し，OS/2を世に送り出したMicrosoftというトップブランドが何となく安心させてくれる．これはまあ，大企業がIBMのメインフレームを選ぶのと同じかもしれない．そして，ブランドイメージだけではなく，WindowsやOS/2などの新しいシステムが登場した場合にも，当然ながらMS-Cの対応が一番早い．この点は，かなりMS-Cを使う際の実質的なメリットだろう．それに加え，品質もよい方だし，そこそこのサポートも受けられる．もちろん価格は0に近ければ近いほどいいのだが，それを別にすればMS-Cは大多数の人がまず納得できる処理系だろうと思う．

新しいバージョンは必ずよいか

どんなソフトウェアでも，バージョンアップされるたびに話題になるのが，どんな点が改善されたのか，改悪ではなくて本当に改善なのかということだろう．ここでは，ごく簡単にMS-Cの各バージョンを外面から比較してみたい．

まず，単純にコンパイラのサイズの面から見てみよう．ver.3.00.17では，コンパイラの各パス(P1.EXE～P3.EXE)の合計は約450Kバイト，ver.4.00(C1.EXE～C3.EXE)の場合にはなぜか約330Kバイトと大幅にシェイプアップされている(ラージモデル用のオプティマイザC3L.EXEを単純に加算しても約450Kバイト)．ver.5.10になると約450Kバイト(C1L.EXEを除く)になり，ver.6.00はIBM-PC版の数字だが，突然640Kバイト(C1L.EXEを除く)にも膨れ上がる．

コンパイルしてできあがる実行ファイルのサイズでいうと，ver.3.00.17で作成した場合にプログラムが一番コンパクトに仕上がるケースが多い．といっても，ver.3.00.17がそれだけ優れた最適化をしていたというわけではなく，ライブラリの仕様が低級で，printf()などのライブラリ関数のオブジェクトサイズが小さかったのが原因である．

また，ver.3.00.17ではライブラリやコンパイラに若干の不具合があったが，『MS-Cハンドブック』(アスキー刊)で指摘されたおかげで(？)ver.4.00では改善されている．ver.4.00は実際，かなり安定したバージョンだったと思う．CodeViewが使えるようになったのも，このバージョンからだった．

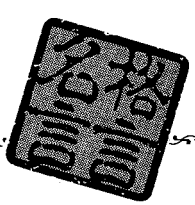
その後OS/2が発表され，OS/2用のアプリケーションやDLLの開発にはMS-C ver.5.*x*以上が必要になる．ver.5.*x*では，「過激なオプティマイズ」をするというので有名で，下手をすると意図したとおりのコードができないという問題(？)があった．ただ，コンパイラの作成するコードのサイズは，一般にver.4.00のときよりさらに小さくなったということは事実である．大まかに言って，MS-Cの場合，新しいバージョンのコンパイラに移行する意味はそれなりにあると思う．

最近発表されたver.6.00には，Programmer's Work Bench(PWB)というプログラム(一種の環境)が付属している．プログラマは，PWBのウィンドウの中からプログラムの編集やコンパイル，実行，デバッグなど，一連の作業が行えるようになっている．まだそれほど使い込んでないので何とも言えないが，

一体誰が使うんだろう

という気がしないでもない．ポップアップメニューを出すときちんと影が付いたり，起動時に前回終了時の状態を再現してくれたりするのは面白いと思うが，プログラマの大多数はPWBを使わずに，これまで通りの無愛想なコマンドラインでのコンパイルを行うのではないだろうか．また，〝cl −help〟とするといきなりウィンドウが現れて，ハイパーテキスト風にヘルプ画面を探索できる．かなり詳細な説明を読めるのだが，これまでの簡潔なヘルプが気に入っていた人にはちょっと面倒かもしれない．

ver.6.00ではANSI標準に対応したほか，いろいろと拡張が施されており，_asm {. . .} によってアセンブラのステートメントを記述できたり，_segmentと_basedキーワードによってfarデータを簡単にアクセスできたり，最適化もプラグマで細かく指定できるようになっている．



残りものには副がある

＜解説＞

関数の引数に渡し切れないパラメータや，関数の戻り値にできなかったパラメータをグローバル変数にしたときの教訓．引数や戻り値自体には注意を払うが，それ以外の変数の振舞いについては，見落しが発生する．そのためその変数の値を操作すると，ときには意図しない副作用を引き起こしてしまう場合がある．

ANSI C なんて大嫌い?

さて，数年前から進められてきたANSIのC言語標準化作業も一段落したようだ．K&R第1版に記載されていない機能のうち，これまで多くのコンパイラが採用してきたenumやvoidなどもANSIの標準で公認された．同様にconstとvolatileが導入されたほか，unsignedに対するキーワードsignedが新設されている．これまでのMS-CはANSIのドラフトを取り入れる形でバージョンアップされてきたので，ver.6.00でANSIの仕様に対応したといっても，これまでのMS-Cの仕様が大幅に変更されたわけではない．それでもANSI C標準との関係で，ver.5.10からver.6.00になった際に変更を受けた部分が何ヶ所かある．

MS-C ver.6.00で変更になった点

まず，符号付き整数と符号なし整数の混在した式の評価方法が変更された．こうした式の場合，ver.5.10まではそれぞれの整数をなるべくunsignedと見なして評価していたが，ver.6.00は値そのものが保存されつつ，評価される．

```
if (-1L < 0xffff)
        /* ver.6.00 */
else
        /* ver.5.10 */
```

上の場合，ver.5.10では−1Lが0xffffffffというunsigned long値と見な

されて比較されるのでこの条件文は偽になるが，ver.6.00 では signed long になるので真になる．

また，switch 文や case の定数で long や unsigned long の値が使えるようになった．ver.5.10 ではコンパイル時にワーニングが出て，long の値が int に切り捨てられていた．

NULL の定義も変更された．これまでは 0x0000(ラージモデルなどでは 0x00000000L)として定義されていたが，ver.6.00 では((void *)0)になった．

このほか，共用体のメンバとしてビットフィールドが使えるようになったり，関数や配列に対して正式にアドレス演算子(&)が使えるようになっている．

さて，シフト演算を行う場合には注意が必要だ．

```
int     i = 1;
long    l;

l = i << 16L;
```

このとき，ver.5.10 では暗黙に i が long に変換されて，結果として l には 0x10000 が代入されていた．ところが ver.6.00 では(というか ANSI 標準では)シフト演算子の左オペランドの型(ここでは int)を保存しなければいけないことになっていて，上の例ではシフトの結果を格納できず，l に 0 が代入されてしまうのである．

関数の定義・宣言

また，関数の定義や宣言のやり方は，昔(K&R 第 1 版の時代)とは随分と違っている．以前なら，次のような形式で関数を定義していた．

```
char *
foo(s, c)
char *s;
int c;
{
...
}
```

ANSIの標準によると，この関数の定義は次のように行う必要がある．

```
char *
foo(char *s, int c)
{
...
}
```

こっちの方がパラメータとパラメータの型を一度に指定できて便利だが，依然としてこれまでのやり方も認められている(MS-C ver.6.00では，−W4オプションを指定すると警告メッセージで「や〜い，古いスタイル」とバカにされるが)．つまり，関数ヘッダは2通りの書き方ができるのである．

関数の宣言も，昔なら次のように書いていた．

```
char *foo();
```

しかし，呼び出すときの引数の数や型が実際の関数定義と一致しているかどうかを厳密にチェックするために，ANSIの標準では次のようにプロトタイプ宣言を使った方法(MS-Cでも以前から採用している)を推奨している．

```
char *foo(char *, int);
```

または,

```
char *foo(char *string, int count);
```

後者の引数名(string と count)には特別な意味はなく，指定しなくてもまったく問題ない．プロトタイプ宣言を行っていない最初の例のような宣言の仕方は将来は廃止する方針のようだが，今のところは警告メッセージも出ない(ANSI で一応公認されているので，出す必要がない)．つまり，3 通りの書き方が認められているわけだ．

このようにいろいろな書き方ができるのを，「C 言語の融通性」だと思う人はあまりいないだろう．もちろん，古い書き方を全部エラーにしてしまうと影響が大きいだろうが，個別の処理系でコンパイラに互換オプションを用意すれば済む話だと思う．みんなが何年も注目していた割には(決して「期待していた」わけではないにせよ)，この辺が中途半端でお茶を濁したような標準になってしまったようだ．こういうのは言語仕様として，

美しくない

と思うが，いかがだろうか．

また，ライブラリ関数も十分な標準化が行われているとは思えない．たとえば，UNIX ではシステムコールに当たる低水準入出力関数(open()や read()，write()関係)についての記述がない．UNIX の場合にはこうした関数は単なるカーネルインターフェイスだから，ANSI の C 標準で決める必要はないのかもしれないが，パーソナルコンピュータ用の処理系ではそうはいかないだろう．今までは UNIX のライブラリが事実上の標準になっていて，たいていの処理系のライブラリはそれを手本に作られていたのだが，ANSI で決められなかった一部のライブラリについては，今後も「UNIX System V 互換」などという風にされるのだろうか．

誰のための標準化か

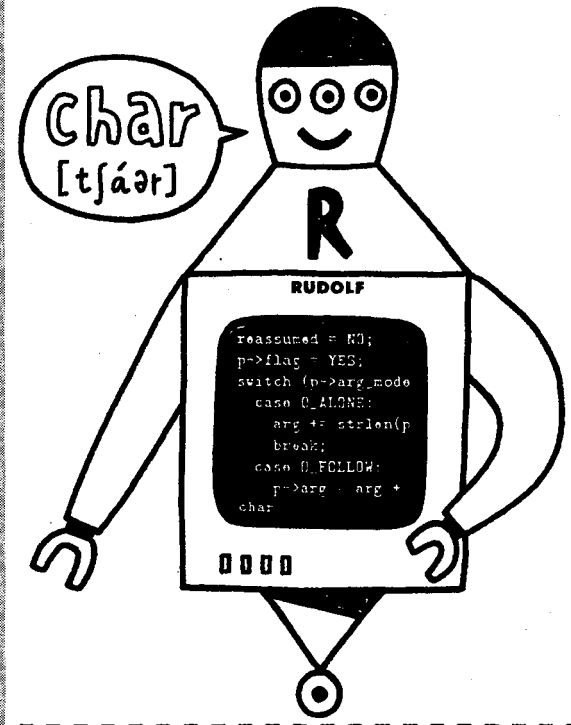
大騒ぎのうちに数年を費やしたANSIの標準だが，できあがってみると現状追認(これはまあ許せるけど)と中途半端の嵐である．こんなことなら，別に標準なんかなくても今のままで一向に困らないような気がする．もっとも，国際機構であるISOと違ってANSIはアメリカ国内の組織じゃないかといちゃもんをつけて，我々は，

NOと言える日本のプログラマ

だぞ，とか強がってみても仕方がないのだが．まあ，標準があれば処理系のメーカーもそれに合わせるように努力するだろうし，プログラマにとってもこれからCを始めようとする人にとっても，まるっきりメリットがないわけではない．

C言語に限らないが，標準化には単に技術的な問題だけではなく，金が絡んでくるからややこしい．自社の仕様が標準に取り入れられれば大きなメリットだし，逆に，標準ができたおかげで独自の機能を持たせたベンダにある程度歯止めがかけられるのだ(その機能なり仕様なりが「事実上の標準」であっても)．まあ標準というのはブランドみたいなもので，確かにユーザーだって「ANSI標準完全準拠」と広告でうたってあれば何となく安心だろうけれど，誰よりも喜ぶのは各社の宣伝担当者だろう．なにしろ今までと違って，「標準」という錦の御旗が自分たちの頭の上にひるがえっているのだから．昔みたいなひどいバグは少なくなったものの，標準準拠だとかって浮かれているばかりでなく，処理系を標準に準拠させる作業の中でわざわざコンパイラにバグを入れるようなことのないように願いたいものだ．

C言語の常識と非常識


Char
[tʃáər]
R
RUDOLF
reassumed = NO;
p->flag = YES;
switch (p->arg_mode
case O_ALONE:
arg += strlen(p
break;
case O_FOLLOW:
p->arg = arg +
char

常識の基準は人それぞれに違っている．それを忘れて，会話の中でついつい「常識」という言葉を使ってしまうため，無用のいさかいが生じるのだ．C言語にも，何種類もの常識が存在する．あるプログラマの常識が，別のプログラマの非常識だったりするのは日常茶飯事だ．この章では，どちらかというとプログラミングの本質部分とはあまり関係のない，Cプログラマのいろいろな常識と非常識を紹介する．さて，この章で述べる常識は，あなたの常識と一致しているだろうか？

C言語には移植性がない

C言語にまつわる誤解にもいろいろあるが，中でも「Cで書いたプログラムは移植性が高い」という勘違いを素朴に信じている人が少なくないようだ．だが，ちょっと考えればわかるように，Cほど柔軟(いい加減)で，必要ならハードウェアに密着した書き方もできる言語でプログラムを書いて，魔法のようにことごとく移植性の高いプログラムができあがるわけがない．一体この勘違いはどこから生じたのだろうか．

まず，「Cで記述されたUNIXというオペレーティングシステムは移植性が高い」という伝説があるが，これは事実である．基本的に80*x*86シリーズのCPU搭載機でしか動作しないMS-DOSなんかと違って，UNIXはそれこそ80*x*86シリーズから68000，RISC，ミニコンからスーパーコンピュータまで，非常に多くのマシンに移植されている(それも比較的短期間で)．アセンブラで書かれるのが当り前のオペレーティングシステムの中で，アセンブラに比べれば高級言語と言えるCで大部分が記述されたUNIXが，群を抜いて移植しやすいのは当然だろう．だからといって，FortranやBASICではなくCで書けばプログラムの移植性が高まるということにはならない．

また，UNIXのCコンパイラは移植性が高い，という事実もある．このCコンパイラは別名pcc(Portable C Compiler)と呼ばれており，lintやyaccを作ったSteve Johnsonという人が作者である．pccが登場するまではRitchieの作ったCコンパイラが使われていたが，RitchieのCコンパイラはPDP-11シリーズ用だった．pccは，その名のとおりコンパイラとしては移植性に優れており，UNIXシステムとともに多くの機種に移植されている．

このような事実がごっちゃになって，いつの間にやら，

Cで書けばプログラムがポータブルになる

という「移植性の神話」ができあがってしまったようなのだ．しかし，Cで移植性の高いプログラムが書けるかどうかは書き方次第，という極めて常識的な事実が存在する以外，実際にはCだからといって特別なことは何もないのである．

移植性だけを取り上げるなら，COBOLやFortranで書いたプログラムの方がずっと優れている．COBOLはDoD(アメリカ国防総省)の音頭で最初の仕様が定められて以後，何回かの改訂を受けながら，最近バカにされつつも相変わらず事務処理分野では圧倒的な勢力を誇っている言語である．Fortranも，世界の計算機の標準言語と言われるくらいに普及しているのは言うまでもないだろう．どちらも古い言語だけに膨大なソフトウェア資産があり，パーソナルコンピュータの世界みたいにCがブームになったからといってPascalから移行するようなわけにはいかないのだ．

Cでプログラムを書くときに陥りやすい移植性の問題点として，ポインタのサイズをついついintと同じに扱ってしまうということがある．実際にはポインタのサイズは処理系依存で，intだとは限らないのだ．ただ，BSDだとintとして扱うのが常識になっているようだから話がややこしいが，少なくとも移植性という点だけを取り上げてみれば，非常に問題がある．

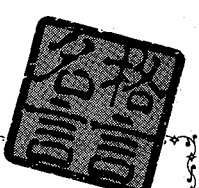
また，Cのライブラリにしても，UNIXのシステムコールインターフェイスとして動作するだけの関数も多い．read()やwrite()などは，Cのライブラリ関数というよりは，本来はカーネルの機能にアクセスするための単なるインターフェイスだし，stat()などの仕様もUNIXのファイルシステム依存になっている．UNIXが標準化されていればまだしも，UNIXに依存したライブラリを使って，移植性が高くなるわけがない．

そして，80x86 用の C の処理系はさらに特殊である．中でも Turbo C は金メダルもので，_ax とか_bx とかのシンボルでレジスタが操作できてしまうのだ．一見便利そうだが，こんなものを使ったら最後，移植性などなくなるのはいうまでもない．

もっとも，

COBOL や Fortran のようになった C なんかいらない

という気もする．マシンに密着した部分まで C で記述できるからこそ，言語仕様上で処理系依存になっているところが多いのだと考えれば，C 言語に移植性がないからといって，がっかりする必要はない．移植する上でどこが問題になるかを把握してプログラミングすれば済むだけの話だろう．



人民の人民による人民のための添字

＜解説＞

マシンのメモリ中では，単に 0 と 1 のビットの列が並んでいるだけ．そこで，人間にもわかりやすい形式でデータの列にアクセスできるように考え出されたのが添字である．ただし，ポインタが苦手だからといって，ポインタを使った方が明快に書ける部分にも添字を濫用すると，かえってわかりにくいプログラムになってしまうので注意．

スタイルからキメてみる

さて，プログラミングが上達してくると，自分のプログラムを他人に評価してもらいたくなるものだ．うまくいけば尊敬されるようになるが，下手すると，

なにこれ，だっせーソース

とか，

動けばいいってもんじゃないだろ

などと，ひどい評価を下されて，えらいショックを受ける羽目になる．将来有望なCプログラマが先輩の心ない一言に傷ついて，「いつか標準化委員会のメンバーになって見返してやる」などとねじ曲がった考えを持たないように，ここを読んでCの正しい(?)常識を身につけるようにして欲しい．

見た目が大事?

さて，プログラマの末路がいろいろであるのと同じように，Cのコーディングスタイルもプログラマによって本当にさまざまである．

本物のプログラマはスタイルにこだわらない

と言われつつ，それでもスタイルにやたらこだわる人が多く，まるで新興宗教のような様相を呈している．

ともかくコーディングスタイルの例をあげてみよう．一番普通なのは，K&R に載っているリストのようなスタイルだ(リスト 2-1)．

```
if ((fp = fopen(*++argv, "r")) == NULL) {
        fprintf(stderr, "Cannot open %s.\n", *argv);
        exit(1);
} else {
        dump_f(fp);
        fclose(fp);
}
```

リスト 2-1　K&R 風のコーディングスタイル

とある現場では，この K&R スタイルが優勢だった．ところがその中にひとりだけ，バークレイボーイと呼ばれる K.K 氏が紛れ込んでいた．K.K 氏は生き延びるため，自分が趣味で書くプログラムは BSD 風に書いてもソースは必ずフロッピーにしまい込み，口うるさい同僚プログラマの目には絶対に触れないようにしていた．そして，書籍に掲載するサンプルプログラムは K&R 風に書いて，迫害を受けないようにしていたのだ．ところがさすがの K.K 氏も徹夜続きで集中力が低下してくると，サンプルプログラムもついつい BSD 風に書いてしまい，周囲の連中に笑われる羽目になってしまったのである．

その BSD スタイルを，リスト 2-2 に示そう．ただし，BSD でもこれは古いスタイルに属するらしく，最近のスタイルはまたちょっと違うという話だ．リスト 2-2 のポイントは，関数名と括弧の間のスペースである．

```
if ((fp = fopen (*++argv, "r")) == NULL) {
     fprintf (stderr, "Cannot open %s.\n", *argv);
     exit (1);
}
else {
     dump_f (fp);
     fclose (fp);
}
```

リスト 2-2　旧 BSD 風のコーディングスタイル

ところがさらに，リスト 2-3 のようなスタイルにこだわる人もいる．

```
if(( fp=fopen( *++argv,"r" ))==NULL )
     {
     fprintf( stderr,"Cannot open %s.\n",*argv );
     exit( 1 );
     }
else
     {
     dump_f( fp );
     fclose( fp );
     }
```

リスト 2-3　マイナーなコーディングスタイル

これはさすがに少数派だが，どの世界でもマイナーはマイナーなりに(であるがゆえに？)意地があるらしく，「〝(〞の後と〝)〞の前にはそれぞれスペースを 1 つずつ入れる」とか「〝,〞の後にはスペースを入れない」というような変な，じゃなくて，一風変わった教義を固守している．

まあ思想信条の自由や宗教の自由は憲法でも保障されているし，どんな

コーディングスタイルを採用するかも個人の自由には違いない．いずれにせよ，せめて読みやすいスタイルにして欲しいものだ．リスト 2-4 のように，インデントがいつのまにかずれてしまって，if と else の対応が一目でわからないような書き方では，バグが混入しやすくなってしまう．

```
if ...
{
      if ...
      {
            ...
}
else {
      ...
      }
}
```

リスト 2-4　if と else のインデントがずれたコーディングスタイル

産みの親は名付け親

プログラムの読みやすさは，インデントのずれがないとか，インデントが何文字分かとかいうようなことだけでは向上しない．他人が読んでもアルゴリズムが理解しやすいようなプログラムを書くようにするのは大原則として，それ以外に，変数名の付け方に気を配るだけでもプログラムの読みやすさは随分向上するものだ．

1 文字や 2 文字の変数名でも，同一の関数内で使われている変数が 2〜3 個ならば，変数名のおかげでプログラムが読みにくくなるというようなことは少ない．が，変数がたくさん使われる場合，たとえば文字列に s1，s2，s3…のような名前の付け方をしていると，わかりにくくて大変である

(変数宣言のところにコメントを入れればよいという問題ではない).

とはいえ，ちょっと極端な例だが〝isRightMouseButtonPressed〟みたいな長い変数名を使えばよいかというと,確かに変数名を見た(というか読んだ)だけで内容はわかるが，これでは昔からのCの簡潔さが損なわれるし，PascalやSmalltalkのプログラムに見えてしまう(だからいいんだという話もある).オブジェクト指向言語によるプログラミングや最近流行のウィンドウ関連のプログラミング(X-WindowやNeXTを見よ)では，なぜか長い変数名が使われる(長いのは変数名に限らない)傾向にあるようだが，いずれにせよCらしさを大事にしたいなら，わかりにくくない程度に短い変数名の方がよいと思う.

ここで参考のために，Microsoftのプログラマ，Charles Simonyiの変数命名法を紹介しよう．この命名法は，Simonyiがハンガリー出身であることから「ハンガリー記法」と呼ばれている.

ハンガリー記法のアイディアは「変数名にその変数の属性を記述する」というもので，基本は

変数の型を示すプレフィクス+識別名

である．プレフィクスは，たとえば〝i〟なら int，〝dw〟なら unsigned long (ダブルワード)を示す．また，そのあとの識別名は，大文字と小文字を混ぜて書く．例を示そう.

```
long    lParam;
DWORD   dwFileSize;
char far *lpszText;
```

最後のlpszTextは,「ASCIIZ文字列へのlongポインタlpszText」という意味になる．ハンガリー記法で使われる主要なプレフィクスを表2-1にまとめておく.

プレフィクス	型
c	char
by	BYTE(unsigned char)
n	short
i	int
b	BOOL(int)
w	WORD(unsigned int)
l	long
dw	DWORD(unsigned long)
s	文字列
sz	ASCIIZ 文字列
p	ポインタ
lp	long(far)ポインタ
np	near ポインタ

表 2-1 ハンガリー記法で使われるプレフィクスの例

こうした記法を使うメリットは，変数の型を変数名に反映させることでプログラムの理解しやすさを高め，無用なバグの混入を未然に避けるというものだ．ハンガリー記法については『プログラミング WINDOWS Volume 1』(アスキー刊)に紹介されているが，『実録！天才プログラマ』(アスキー刊)では Simonyi 自身がこれについて語っているので，興味があれば読んでみて欲しい．

関数に気を使う

プログラミングの際にけっこう頭を悩ますのが，関数名の付け方である．本気でかっこいい関数名を付けようと思ったら，やはり英語の素養が必要になるだろう．英語がどうもあまり得意でないなら，せめてわかりやすさを重視すべきだ．その場合，C のライブラリ関数の命名法が参考になる．何かをチェックして TRUE か FALSE を返すなら「is～」，変換なら「～to～」(または「to～」，「～2～」)，取得なら「get～」，書き出しなら「put～」，文字列関係なら「str～」というように，ある程度の法則性があるし，語彙も

多くない．また，連結(concatenate)なら「cat」，比較(compare)なら「cmp」というような略記法もほぼ確立されているので，ライブラリ関数の一覧を研究して応用するとよい．

関数名も大事だが，ソースプログラム中の関数の位置にも気を配る必要がある．main()を含め，ソース中で関数の階層構造(下請け・元請け関係)がある程度はっきりするように書かないと，いくらかっこいい関数名を付けてもバカにされるのが落ちである．その場合，main()がソースの先頭にくればトップダウンのソースになるし，最後にくればボトムアップのソースになる．前者にはソースを先頭から読んでいけるというメリットはあるが，main()などの上位の関数内で呼び出している下位の関数の型宣言がいちいち必要になってくる．また，ソースの読みやすさを向上させるには，関数のスコープにも配慮して欲しい．あるソースファイルの内部だけで下請け的に使われる関数には必ずstaticを付けるようにすると，それだけでもわかりやすさがかなり違ってくるものなのだ．

コメントにまで気を使う

コメントの付け方というのもけっこう難しいものだ．どんなコメントをどの程度付ければよいかは，プログラムのあり方(自分だけがわかればよい，他人がメンテナンスする可能性がある，書籍に掲載する，など)にもよるので一概には言えないかも知れない．

「1行いくら」でプログラム開発を請け負ったまではよかった(?)ものの，幸か不幸か有能なプログラマに育ったばっかりに，非常に簡潔なソースができあがってしまったとしよう．これでは自分の才能と支払われる報酬が見合わないと思った場合，せっせとコメントを入れて行数稼ぎをしたくなる．それならそれでやりようがあるわけで，プログラムの仕様書や変更ログ(RCSなどのソースコード管理プログラムを使って出力するのが一般的)を全部ソースに入れてしまえばよいだろう．

しかし，コメントはやはり「保守性を向上させる」ために入れるのが普通だ．とはいうものの，いくら美しいコメントを入れたって，コメントを入れなければわからないようなプログラムでは話にならない．何よりもわかりやすいコードを書くのが基本中の基本である．

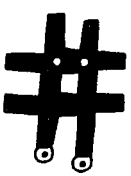
それでも適度にコメントが入っていれば，プログラムの読みやすさがかなり向上するのも事実だ．ソースファイルの先頭や，各関数の前にコメントを入れておく場合が多い．

ソースファイルの先頭に入れるコメントには，ファイル名や作者名，変更ログ，バージョンなどを書いておくのが一般的なようだ(リスト 2-5)．

```
/*
 *   foo.c
 *      by A.Y
 *              ver.1.0 1990.1.4  initial version
 *              ver.1.1 1990.2.5  -f option
 *              ver.1.2 1990.2.22 Kanji support
 *              ver.1.3 1990.2.24 -i option
 */
```

リスト 2-5　ファイルの先頭にあるコメントの例

RCS などのソースコード管理システムを使っている場合には，変更ログをファイルの末尾にまとめて入れておいてもよいだろう．



関数の前には，関数の目的や引数について簡単に書いておく(リスト2-6).

```
/*
 *  int readconfig(FILE *fp)
 *       - Read the configuration file. If
 *       successful,return 0; otherwise,
 *       return -1.
 *  FILE *fp - Pointer to the configuration file.
 */
```

リスト2-6　関数の前にあるコメントの例

だが，こういうコメントを関数ごとに書くよりも，仕様書を別にきちんと書く方がよいかも知れない．確かにソースファイルと仕様書が別々になっているより，こうしたコメントをソースファイルに入れておくと管理は多少楽になるのは事実なのだが．そのうち，ひょっとしたらMS-CのPWBでもソースと仕様書がハイパーテキスト風に管理されるようになるかも知れない(関数定義をクリックすると仕様書が読めるとか).

このほかにも，プログラム中で重要な部分や，if文やループのネストが深いところに簡単なコメントを付けておくとよいだろう．

また，プログラム中の古い部分，もう使われなくなった部分をコメントアウトしてしまいたい場合も多々あるはずだ．このとき，コメントにするつもりで単純に〝/*～*/〟を使うと，コメントがネストしてしまってコンパイル時にエラーになってうまくいかなかった経験があるだろう．そういう場合には，次のように「#ifdef notdefの技」を使えばよい．

```
#ifdef notdef
        …古いコード…   /* comment */
#endif
```

notdef などというシンボルを#defineすることはまずありえないので，これでうまく古いコードを殺してしまえるのだ．〝#ifdef notdef〟の代わりに，〝#if 0〟を使う人もいる．

さて，「コメントはわかりやすさが第一」なのは当然だが，コメントに果たして日本語を使うべきかどうかというローカルな論争がよく行われる．日本語コメント賛成派は，「英文は何とか読めはするが，自分では書けない」，「書くにも読むにも日本語が一番わかりやすい」という，一応もっともな主張をしている．これに対して反対派は，「漢字が使えないエディタもある」，「外国人がデバッグするのに不便」，「どんなシステムでも日本語が通ると思ったら大間違いで，MS-DOSから別のシステムに移植するときに大変」，「16ドット×16ドットの端末で漢字混じりのプログラムを眺めると汚いソースに見えちゃう」という主張をもって対抗している．確かにMS-DOSとMS-Cを使ってプログラミングしている限り，どうしても日本語のコメントは避けるべきだという主張にあまり説得力はないかも知れないが，基本的に英数字や記号ばかりのプログラムの中に日本語が混じっているのは嫌だな～という気がする．さて読者の皆さんはどう判断されるだろうか．

スペルミスは恥しいぞ

コメントの問題が出たついでに，スペルミスについて簡単に触れておこう．

以前，アメリカ人と仕事したときのことだ．その人は，何かの本に掲載されていたプログラムを見てすぐに変数名だかコメントだかの英語のスペルミスを発見すると，

コ～ユ～ノハ一番笑ワレマス

と(日本語で)いったのである．そのときは「へえ，そういうものか」と思

ったが，確かにどんなに優れたプログラムでもエラーメッセージにスペルミスがあったりすると笑ってしまうし，これが腐ったプログラムでのスペルミスなら，笑うどころか納得してしまうだろう．

〝igunore〟(ignore)とか〝handel〟(handle)とかのみっともないスペルミスを避けるためには，正しいスペルを知らなければ話にならないが，タイプミスにも十分気を付ける必要がある．たとえば，〝return〟の綴りを知らない人はあまりいないと思うが，なぜかこれを〝retrun〟などとタイプミスしてしまうケースが多い．ソースプログラム中で，

```
retrun  (-1);
```

などと書いてあると，この間違いがコンパイル時に検出されないから厄介である(関数retrun()の呼び出しだと判断されてしまう)．この教訓から，

returnの値は括弧でくくるな

というCの格言が生まれた(?)のだ．

さて，コメントやシンボルでのこうしたスペルミスを検出してくれるスペルチェッカがないものかと誰しも思うだろう．そんな中で，MS-C ver. 6.00のPWBでは，MS-WORDのスペルチェッカと連動する機能があるらしい．まだ試す機会がないのだが，これを使えばある程度のスペルミスは防止できるようになるかも知れない．

関数の行数にまでこだわる

Cのプログラムを書くのに，いくらなんでも数千行ものmain()1つだけで済ます人は(多分)いないわけで，たいていはあまり長くなり過ぎないように，プログラムの構造を考えつつ，いくつかの関数に分割して書く．

ところがやっぱり関数の行数にもこだわる人がいて，あるプログラマは「1関数は40行まで」というポリシーを持っていた．実はこれ，IBM-PCのEGA(表示アダプタ)の1画面43行表示モードを使っていたからで，昔はこの人PC-98XAを使っていたので「1関数30行まで」がポリシーだったのだ．最近はIBM-PCのVGAを使っているので，「1関数50行まで」にしているという．これに対して，

10行以下か，500行以上

をモットーにしている(?)プログラマもいたりして，関数の行数1つとってもプログラマの個性やこだわりが発揮されている．

プログラムを読む

優れたソースプログラムを読むのは，C言語上達の1つの方法だと言われる．しかし，見るからに読みにくいプログラムなど，無理して読む必要はない．そんなのはろくでもないプログラムに決まっているからだ(すごい偏見)．ただ，誤解しないで欲しいのは，プログラムの複雑さと読みにくさは別だということである．読みにくいプログラムというのにもいろいろ考え方はあるだろうが，アルゴリズムが読み取りにくいとか，typedefすればすっきりするのにわざわざ複雑なやり方で定義してみたり，if～else if～などのインデントがめちゃくちゃで解読に苦労したりというようなことを思い浮かべていただければよい．

さて，ここでは「Cプログラムの音読法」を取り上げよう．なにそれ，と思われるかも知れないが，リストを打ち込んで2人で読み合わせをする場合だってけっこうあるだろうし，それに英語を習い始めの頃，

教科書を声に出して何度も読むように

と先生から言われたはずだ．声に出して読めば頭にも入りやすくなる．また，自分の読み方は果たして正しいのか，という不安を抱いたままでは，同僚のプログラマともコミュニケーションがとりにくくなってしまう．何よりも，プログラミングが芸術で，プログラムが作品ならば，詩人が自作の詩を朗読するように，

プログラマが自分のプログラムを朗読したっていいではないか

日本人なのに英語の歌を歌う時代なのだから，そのうちC言語のプログラムや16進ダンプをメロディーにのせて歌うアイドルやバンドがデビューする可能性だってないとは言えない(そういえば中山美穂のデビュー曲は『C』だった)．

そこで，Cプログラムで使われる記号，キーワード類の読み方をまとめてみることにした．ここで示すのは我々の読み方なので，現場や，また文脈によっても読み方が違うかも知れない(実際，あちこちで周期的に論争が起こっている)が，いちいち文句を言わないように．

記号	読み方	その他の読み方
␣	スペース	アキ
!	ビックリ	エクスクラメーション，バン
"	ダブルクォート	
#	イゲタ	シャープ，ナンバー
$	ダラー	ドル
%	パーセント	
&	アンパサンド，アンド	
'	シングルクォート	アポストロフィ
`	バッククォート	
(	カッコ	パー，開きカッコ，丸カッコ
)	コッカ	レン，閉じカッコ，丸カッコ閉じ
*	アスタ	スター，星，アステリスク
+	プラ，プラス，足す	
,	カンマ	コンマ
−	マイナス，ハイフン，引く	マイナ
.	点，ピリ	ポツ，ドット
/	スラ	スラッシュ
:	コロン	
;	セミコロン	
<	小なり	レスザン
>	大なり	グレーターザン
=	イコール	
?	クエスチョンマーク	ハテナ
@	アットマーク，アット	
[	大カッコ，カギカッコ	ブラ，角カッコ
]	大コッカ，カギコッカ	ケット，大カッコ閉じ，角カッコ閉じ
＼	バックスラ	バックスラッシュ，逆スラ
^	ハット	サーカムフレックス
{	中カッコ	波カッコ
}	中コッカ	中カッコ閉じ，波コッカ
~	ニョロ	チルダ，チルド
_	アンダースコア	アンダーバー，下線
|	縦棒	パイプ記号，バーチカルバー

表 2-2　記号の読み方

キーワード	読み方	その他の読み方
auto	オート	
break	ブレーク	
case	ケース	
char	チャー	キャラ，キャラクタ
const	コンスト	
continue	コンティニュー	
default	デフォルト	
do	ドゥー	
double	ダブル	
else	エルス	
enum	イナム	エナム
extern	エクスターン	
float	フロート	
for	フォー	
goto	ゴートゥー	
if	イフ	
int	イント	インテ，インテジャ
long	ロング	
register	レジスタ	
return	リターン	
short	ショート	
signed	サインド	
sizeof	サイズオブ	
static	スタティック	
struct	ストラクト	
switch	スイッチ	
typedef	タイプデフ	タイプデファイン
union	ユニオン	うにおん
unsigned	アンサインド	
void	ボイド	
volatile	ボラタイル，ボラティル	バラトル
while	ホワイル	

表 2-3　C の予約語の読み方

関数名	読み方	その他の読み方
fopen	エフオープン	
printf	プリントエフ	
scanf	スキャンエフ	スキャンフ
getc	ゲットシー	
gets	ゲットエス	ゲッツ
puts	プットエス	プッツ
getchar	ゲットチャー	ゲットキャラ
strcpy	ストラコピー	エスティーアールコピー，ストリングコピー
strcmp	ストラコンプ	エスティーアールコンプ，ストリングコンプ
strtok	ストラトーク	エスティーアールトークン
strlen	ストラレン	エスティーアールレン
strncat	ストラエヌキャット	エスティーアールエヌキャット
memcpy	メムコピー	
malloc	エムアロック	マロック
calloc	シーアロック	キャロック
atoi	エートゥーアイ	アトゥーイ，アトイ
atol	エートゥーエル	アトゥール，アトル

表 2-4　C の標準関数の読み方

その他	読み方	その他の読み方
stdio	エスティーディーアイオー	スタッドアイオー，標準アイオー
stdin	エスティーディーイン	スタッドイン，標準入力
EOF	イーオーエフ	エンドオブファイル
NULL	ヌル，ナル	
io.h	アイオーエイチ	アイオーテンエッチ

表 2-5　その他のキーワードの読み方

コンパイルエラーの調理法

ある日，山田君(仮名)は自分のプログラムのデバッグをしていた．ところが，いくらソースを直してプログラムを実行してみても，ちっともバグが潰れないのである．おかしい．変だ．そのうち同僚のプログラマが集まり始めて一緒に考えてくれたが，ソースを見ても誰も原因がわからない．コンパイルー実行を繰り返しながら，ひょっとしたらコンパイラのバグだろうかと山田君(仮名)が思い始めたそのとき，画面を見ていた別のプログラマがポツンとつぶやいた．

あれ，何でCLに-cオプション付けてるの?

そう，山田君(仮名)はMAKEを使っていなかったために，バチが当たったのである(「いくらコンパイルしても実行ファイルが更新されていなかった事件」)．

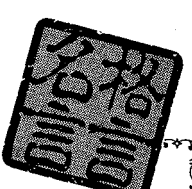
さて，人間の習性に起因するこういう間違いを直すのは難しいが，コンパイルエラーに対応するのは慣れればさほど大変ではない．

まず，ワーニングにはどう対処するか．コンパイル時にワーニングが出ても，別に動作に支障がない場合も多い．これをそのままにしても問題のないこともあるが，やはりワーニングとはいえ，きっちり処置しておきたいものだ．ワーニングが出ないようにプログラムを再チェックすればデバッグと同じ効果があり，よりエレガントなプログラムにできるだろう．また，1つのプログラムを徹底的に改良することで，プログラマとしての自己訓練の効果もあるはずだ．型変換については，自動的に型変換される部分を自分で把握してきちんと管理し，必要に応じてキャストするようにすれば，コンパイル時に-W3を指定しても，ワーニングはほとんど出ないようになる．必要があってどうしてもワーニングを避けられない部分には，コメントを書いておくとよいだろう．これはまあ自己満足といえなくもない

が，他人がそのソースを見たときに，ひょっとしたら「う〜ん，さすが」と思ってくれるかも知れない(?)．

また，一見するとよくわからないエラーが出る場合がある．〝;〟がないと怒られて，指摘された部分のソースの見ても問題がない，あるいは，EOFになっちゃったと文句を言われたような場合には，〝}〟がどこかで抜けていることが多いものだ．これはちょっと厄介で，使っているエディタに括弧の対応を調べる機能(viなら%)がなければ，ソースを注意深く読んでいくほかはない．

#endifの抜けも厄介である．#ifdef〜#endifがソースファイルの中で何ヶ所にも使われていたり，何重にもネストしていたりすると，どこで#endifが抜けているのかを見つけるのが大変なのだ．そういう場合にはソースをあちこち眺めて調べるよりも，〝cl −P〟でプリプロセッサの出力をファイルに落として調べると効率的である．



溺れるものは笑われる

＜解説＞

通ぶってわざと複雑な式を書いたりすると，素直に書いた式よりも効率の悪いコードをコンパイラが生成する場合がある．このように，自分の生半可な技量に溺れて見栄でコーディングすると，優秀なプログラマに笑われる羽目になるという教訓．

Cを使う覚悟

さて，これまでC言語にまつわるいろいろな話(単なるヨタ話だという説もある)をしてきたが，キーワードの読み方1つをとってみても，Cプログラマにはやたらとこだわる人が多かったり流儀があったりする．読者がCのプログラマを目指すなら，関数の行数が多いだの少ないだの，関数名や変数名がダサいだの，「charはキャラじゃなくてチャーだ」だの，「こんなとこにスペース入れるなんてバカじゃないの」，「コメントぐらい英語で書けるようにしろよな」等々，先輩プログラマのありとあらゆる悪口雑言に耐え抜く覚悟が必要になる．おまけに，Cのプログラムには移植性がないと思った方がよいときているのだ．

ああ，こんなことならFortran屋になればよかった

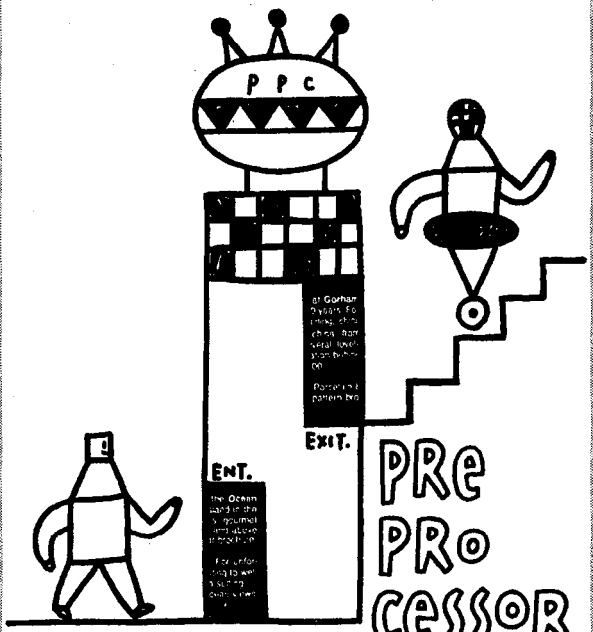
とか，

これからはサラリーマン相手に
COBOL屋として再デビューしよう

と思ったあなた．正解です．1日も早くやくざな世界から足を洗って，清く正しい標準への道を歩んでください．

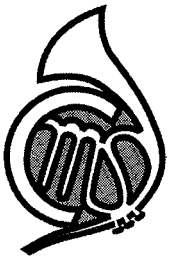
本書のここから先は，それでもめげずにヘラヘラ笑ってCプログラマになろうという無茶な人のために書かれている．この本を読み終わったとき，あなたがCウィザードになれる自信を身に付けているか，それとも単なる社会不適応者になっているかは神のみぞ知ることだろう．

プリプロセッサ道場



どの世界にも，重要なのに何となく地味，という損なイメージを持たれているものがある．会社でいえば，さしずめ総務部みたいなものだろうか．「うちの会社はなんといっても営業がメインだから」とか「開発スタッフが主役ですね」ということはあっても，なぜか「我が社のスターは総務部です」と言うような会社はない．映画の無責任男やスチャラカ社員は，総務部なんかで働かないのである．裏方という仕事柄，これは仕方のないことかも知れないが，元総務部社員としては少々残念な気がする．

Cのプリプロセッサも，確かにあまり派手な存在とはいえない．だが，マクロをうまく使いこなせば，Cプログラムの書きやすさや質を大幅に向上させられるのだ．プログラムがちょっと書けるようになったからって，Cをマスターした気になるのは100万年早い．気のきいたマクロが書けるようになることが，ウィザードに成長するための必須条件なのだ．



マクロ定義の光と影

文字列を扱うプログラムを書いていると，必ずと言っていいほど文字種別の判断を行う必要がある．たとえばある文字が数字かどうか，アルファベットかどうかなどだ．その評価関数であるisdigit()やisalpha()などはどう実現されているだろうか．マクロの話の手始めとして，よく取り上げられる文字種の判別の問題を再検討してみよう．

単純に考えると，文字種の判別にはリスト3-1のようなマクロを定義してやればよいように思える．

```
#define isdigit(c)      ((c) >= '0' && (c) <= '9')
#define isalpha(c)      (islower(c) || isupper(c))
#define islower(c)      ((c) >= 'a' && (c) <= 'z')
#define isupper(c)      ((c) >= 'A' && (c) <= 'Z')
```

リスト3-1　最も単純な文字種の判別

ところが，これには次のような問題点があるのだ．

問題点1：引数の評価を2度以上行っている
問題点2：特定の文字種のコードが連続していることが前提
問題点3：何度も比較を行うのは遅い

この節では，上記の文字種判別マクロにまつわる3つの問題点を掘り下げていくことによって，より優れたマクロを定義するための留意点を明らかにしていく．

副作用に気をつけろ

まず，第1の問題点を探ってみよう．なぜ引数の評価を2度以上行うのが問題になるのかというと，このマクロが，

いかにも関数みたいな形をしている

ということに起因している．そのため，関数のようなつもりでうっかり引数に副作用のある式を書くと，予期しない副作用が発生してしまうのだ．たとえば，上記のようなisalphaマクロを，

```
while (isalpha(*s++))
```

などと使おうものなら，最悪の場合，sは各繰り返しごとに4回インクリメントされることになる．そしてこの手のバグは慣れないとなかなか発見しにくいのである．

読者諸兄のように良識のある方々は，関数の引数に副作用のある式を書いたりしないだろうが，お友達はどうだか分ったもんじゃない．他人が利用しそうなプログラムには特別の注意を払わなければならないのだ．

お前の書いたマクロのせいで納期に遅れた

などと言われないように，少しぐらい(実行速度が)遅くなっても関数にしておくのが無難だろう．お友達に悪癖があったとしても，見なかったことにしておく口実にもなる．煩わしいことは避けて通るに越したことはない．

コード体系に依存するな

2番目にあげたコードの連続性の問題はちょっと想像し難いかも知れないが，大変重要なことなのだ．たとえばisupperに注目してみよう．このマクロは，先ほどの定義のやり方では，

文字のコードが'A'の文字コード以上であり，
かつ'Z'の文字コード以下である

ということをチェックしている．

幸いなことに，広く用いられているASCII(社名ではなく，American Standard Cord for Information Interchange)のコード体系では，'A'は0x41で，'B'が0x42，この調子で'Z'が0x5aというように，'A'から'Z'までには連続したコードが割り当てられている．だからisupperマクロのチェックに「ひっかかれ」ば，その文字は間違いなく「大文字」なのだ．

しかし，しかしである．「CはASCII(社名ではない)のコードを使用する」とは少しも決められていない．文字コードに関しては，

実行時のマシンの内部コードを使用する

のである．世の中に存在する文字コードはASCII(社名ではない)のコードばかりではなく，それどころか，世の中で最も広く用いられているのが，「不連続点」を持つEBCDIC(Extended Binary-Coded Decimal Interchange Code)なのである．ちなみにEBCDICだと，アルファベットの，'I'のコードは0xC9で，'J'のコードは0xD1である．文字の大小関係('A'は'Z'より前にある)は成立するが，連続しているとは限らないのである(数字のコードは連続している)．

そうなると先のisalphaのマクロは，EBCDICのマシンでコンパイルす

ると危険をはらんだコードが生成される．本気で移植性を考えるなら，偏執的と言われようが何だろうが，

```
#isalpha(c)      ( (c) == 'A' || (c) == 'B' || \
                                (c) == 'C' ... (c) == 'Z' )
```

と書くべきだろう．関数にした方がもっといいかもしれない．

ところで，文字コードの違いによって移植性が失われる場合があることが明らかになったが，実をいうと，ここまで考える必要はない．なぜならば，ライブラリがあるからだ．

〝ctype.h〟をインクルードすると，isdigit や toupper など文字コードに依存した処理を行うマクロ(あるいは関数)が使用できるようになる．このようにライブラリとして供給されるものは，先人の手によって，処理系依存の問題は解決されているのだ．だから，こうしたライブラリマクロ(あるいは関数)を積極的に使うべきだろう．ASCII(社名ではない)コードのマシンでも，EBCDIC のマシンでも isupper は isupper として正しく機能する(少なくとも，機能する可能性が高い)．

それでも，自分のヘッダーファイルで定義すれば，その分だけ行数が増えて実入りが大きい(？)ように思うかもしれないが，問題が発生したとき

自分のバグではありません．ライブラリの仕様です

という伝家の宝刀をふるうことができなくなる．どちらを選ぶかは読者の好みだ．もっとも世の中には，ライブラリによる問題が発生しても，責任をもって対処してくれないコンパイラメーカーも多いから注意が必要だ．

比較を 1 回で済ます

何度も比較するために遅くなるという 3 番目の問題点は，マクロの定義内容をみれば一目瞭然だろう．アルファベットであるかどうかを判定するにしても，4 回の比較を行わなければならないのだ．そしてその度に引数の評価が起こる．

```
if (isalpha(argv[i][j])) ...
```

先の isalpha のマクロの場合には，上の if 文を評価する際にインデックスの計算が 4 回起こり得る(今のコンパイラは最適化が進んでいるので，実際には，1 回で済むかもしれない)．isalpha マクロの定義自体がそもそも 4 回の比較を必要としているので，これは根源的な問題だといえるだろう．なんとか 1 回の比較で，文字種別を判断することはできないだろうか．

そこで一般のライブラリで使われているテクニックを紹介しよう．それは，文字種別のテーブルを用意する方法である．テーブルへのインデックスとして文字コードを用い，テーブルの各エントリは 8 ビットで，それぞれのビットが文字種に対応している(図 3-1)．

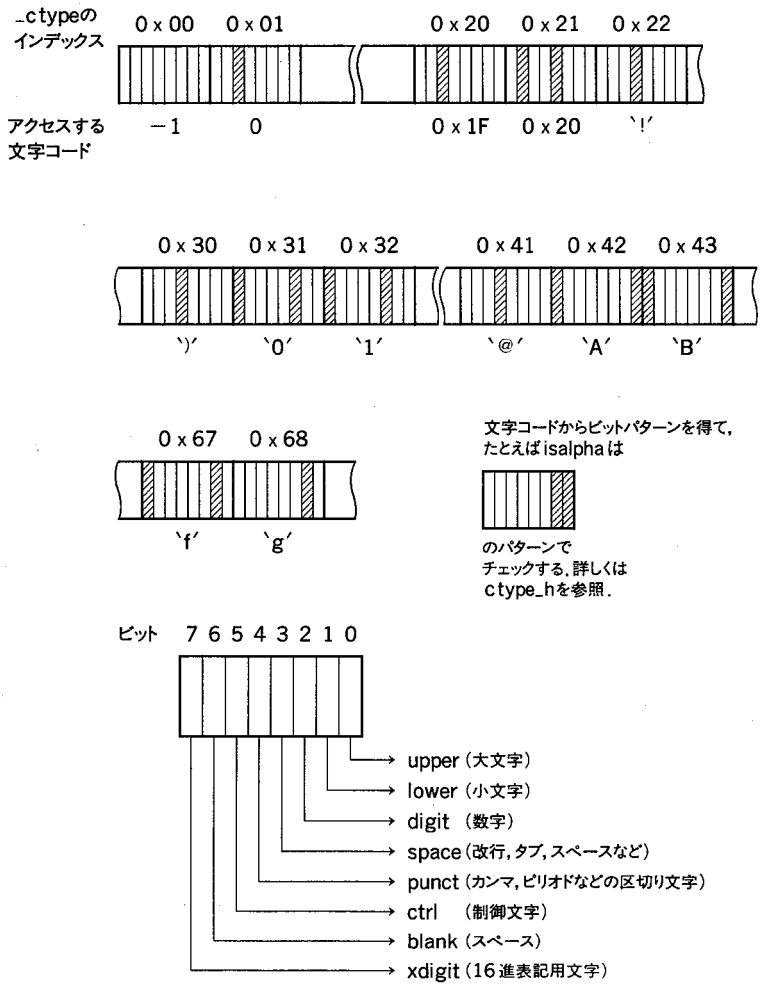


図3-1 _ctypeのインデックスとアクセスする文字コード

たとえばMS-Cのライブラリだと，リスト3-2のようになっている．

```
extern unsigned char _NEAR _CDECL _ctype[];

#define _UPPER  0x1     /* アルファベット大文字 */
#define _LOWER  0x2     /* アルファベット小文字 */
#define _DIGIT  0x4     /* 0～9の数字 */
                        .
                        .

#define isalpha(c)      ( (_ctype+1)[c] & \
                                (_UPPER|_LOWER) )
#define isupper(c)      ( (_ctype+1)[c] & _UPPER )
#define islower(c)      ( (_ctype+1)[c] & _LOWER )
#define isdigit(c)      ( (_ctype+1)[c] & _DIGIT )
                        .
                        .
```

リスト3-2　MS-Cでのマクロ定義

このやり方だと，アルファベットかどうかの判定にも，引数の評価は1回で済む．前の例ではisalphaがisupperとislowerの式の論理和であったのが，ビットの論理和になっているのが，なんとも心憎い点だ．

さて，ここで1つ注目してもらいたい．テーブルを参照する際に，なぜ，

```
_ctype[c]
```

ではなく

```
(_ctype+1)[c]
```

なのか，お分りだろうか．それはis～マクロの引数として－1を与えても，

ちゃんと_ctypeの中のアクセスで済むようするためなのだ．−1はgetchar()などのEOFとして返されることを思い出してもらいたい．そのため，うっかりしたプログラマがEOFであるかどうかのチェックをする前に，is〜マクロを呼んでしまうことに備えているのである．

ここまで考えてくれてるなんて・・・

と，思わず目頭が熱くなるのは私だけではあるまい．

また，テーブルによる解決策は，先の問題点1(式を2回評価する)も解決している．そして問題点2のコードの連続性という前提すら不要である．もしも違ったコード体系のテキストを扱おうとしたときには，再コンパイルの必要はなく，テーブルの差替え，つまり再リンクを行えば良いだけなのだ．

ところが，この_ctypeテーブルには，またしても移植性についての問題がある．米国では，文字コードとしてASCII(社名ではない)を使用しているが，このコード体系は，0から127のコードしか規定していない．そのため，米国仕様のライブラリは，_ctypeテーブルの大きさは129エントリである．

しかし日本ではASCII(社名ではない)のコードを拡張したJISコードを用いていて，これは0〜255のコードを規定している．そのため，日本語仕様のライブラリは，_ctypeテーブルの大きさは257エントリなのである．

日本の環境でプログラムを書く場合には，charはunsignedとして使い，そのunsigned char値を使用して，ほとんど無意識にis〜マクロをアクセスしている．しかしそのプログラムをなまじ米国仕様のコンパイラでコンパイルして，日本語のテキストを食わせたりすると，正しく動かないプログラムが簡単に作成できてしまう．そこで，移植性を本気で考える人は，is〜マクロで評価する前に，isasciiを用いて，ASCII(社名ではない)のコードであるかどうかを検査するべきだろう(isasciiは引数をunsigned intにキャストし，0x80と比較する)．

嘘のような本当の落とし穴

さて，ここでちょっと別な例を取り上げよう．たとえばデバッグ時に情報を表示したい場合などに，次のようなマクロを書いたりしていないだろうか．

```
#define debug(s)      fprintf(stderr, \
                              "%s\n", (s))
```

だが，このマクロには強烈な落とし穴がある．百聞は一見にしかず，次の例を見てもらいたい．

```
debug("It's just like rock'n'roll.");
debug(message[0]);
```

これをある処理系に展開させると，なんと以下のようになってしまう．

```
fprintf( stderr,
        "%"It's just like rock'n'roll."\n",
        ("It's just like rock'n'roll.") );
fprintf(stderr, "%message[0]\n", (message[0]));
```

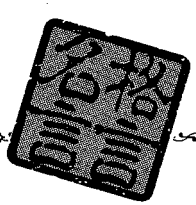
これは，プリプロセッサがfprintfのフォーマット文字列中のsをトークンと解釈し，展開してしまったためなのだ(幸いMS-Cではこのようなことはない)．

このように文字列の中でまでマクロの展開を行うというのは，文字列化演算子('#')とトークン結合演算子('##')が登場したおかげで最近は必要なくなった．文字列の中のたまたま同じ綴りをトークンと解釈して展開するのは，文字列化演算子のない処理系にありがちである．

このdebugマクロをプリプロセッサが誤解しないようにするには，次のように書く必要があったのだ．

```
#define debug(str)    fprintf(stderr, "%s\n", str)
```

トークン結合演算子とプリプロセッサの機能については，あとでまた話をしよう．



百文は一呼出しにしかず

＜解説＞

プログラムの分かりにくい部分に長い（分かりにくい）コメントをだらだらと書くよりも，別関数にして，きれいに書き直した方がよいこと．あるいは自分の書いた関数の性能がライブラリ関数の呼び出しとたいして変らないときを嘆いたようす．

マクロ・マクロ・マクロ!

この節ではマクロの使い方を紹介する．マクロの本来の用途は定数や数式のシンボル化なのだが，「便利なマクロ」というものを求めようとすると，ただの置き換えというよりも，コーディングテクや(レ？)トリックを隠したり，分りやすくするための使命を帯びたものになる．

たかが置き換え，されど置き換え

マクロの基本的な機能である式の置き換えだけでも，結構複雑なもの(あるいは便利なもの)を実現できる．のっけからややこしい例で恐縮だが，ビット操作をやってみよう(256の意地と言われるほど複雑なものでもない)．

配列というのはすごく便利なデータオブジェクトだが，その最小単位がcharであるというのが「もったいなくて」気に入らない．なんとか4ビット値の配列や，2ビット値の配列を実現できないだろうか．そこで考えたのがリスト3-3のマクロである．

```
#define retrieve4(tbl,i) \
    ((tbl[(i)/2] >> (4 * ((i) % 2))) & 15)
#define store4(tbl,i, n)     \
    ( tbl[(i)/2] = tbl[(i)/2] \
        & ~(15 << (4 *((i) % 2))) \
        | (((n) & 15) << (4 *((i) % 2))) )
```

リスト3-3　4ビットの配列を実現するマクロ

上からそれぞれ，配列からの取り出し，配列への格納だ．簡単のために char は8ビットだとしておく．目的の4ビットは char の配列のちょうど半分のインデックスの上位か下位の4ビットにある．4ビット配列のインデックスが奇数が偶数かににによって，シフトしてやればいい．そして，その4ビット値をマスクするのが「15(0xF)」だということは自明だ．図3-2も書いてみたので，そちらも参照して欲しい．

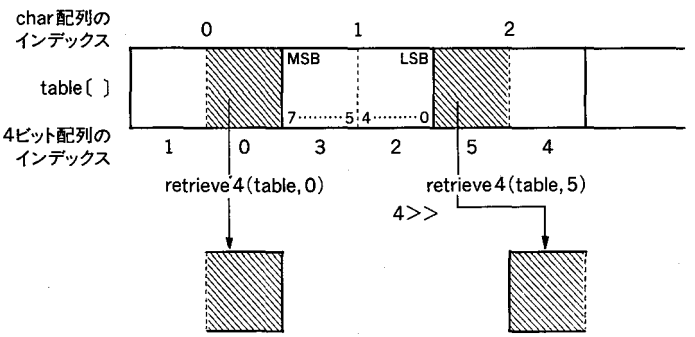


図3-2　4ビット配列の考え方

いまの例では，マクロに数値定数をそのまま書き込んでいる．具体的に考えた方が分りやすい場合もあるからだ．我々プログラマの頭の中は，「本物の代数屋」ほど抽象的にすっきりしているとは限らない．

この泥臭いマクロを少しばかり格好よくしよう．格好よくするには，定数や部分式の隠された意味をみつけて，シンボル化するのがよい．まず定数の意味から考えよう．実はこのマクロは簡単に2ビット配列のマクロに変更できるのだ．リスト3-4がそれだ．

```
#define retrieve2(tbl,i)     \
    ((tbl[(i)/4] >> (2 * ((i) % 4))) & 3)
#define store2(tbl,i, n)     \
```

```
( tbl[(i)/4] = tbl[(i)/4] \
    & ~(3 << (2 *((i) % 4))) \
    | (((n) & 3) << (2 *((i) % 4))) )
```

リスト3-4　2ビットのやつ

このマクロはもとの4ビット配列のマクロの定数を，

Cプログラマの本能

のおもむくままに変更したものである．このマクロを「じっと見る」と正しいのでほっとする．「本能」を見直したりしそうだが，冷静に確認しなければ，それは「本能のまま」のプログラムである．

具体例が2つもあれば，「魔法の数字」や部分式の意味が分かってくる．そこでこれらをシンボル化しよう(リスト3-5)．

```
#define BWIDTH      4    /* for 4 bit array */
#define CELLWIDTH   8    /* hard implementation */
#define _MB_        (CELLWIDTH / BWIDTH)
#define _VM_        ((1 << BWIDTH) - 1)

#define _bshift(i)  (BWIDTH * ((i) % _MB_))
#define retrieve(tbl,i) \
    ( (tbl[(i)/_MB_] >> _bshift(i)) & _VM_ )
#define store(tbl,i,n) \
    ( tbl[(i)/_MB_] = tbl[(i)/_MB_] \
        & ~(_VM_ << _bshift(i)) \
        | (((n) & _VM_) << _bshift(i)) )
#define entries(n) \
    (((n) + (_MB_ - 1)) / _MB_)
```

リスト3−5　シンボル化したやつ

ページ数の都合で,「ちょっと魔法」なシンボルを使ってあるが,読者はより快適なシンボルを考えてみて欲しい.もしも余裕があれば,符号付きの値を扱えるようにもしてみて欲しい.

こうしてシンボル化すると,BWIDTH が 1 であれば,1 ビットの配列も実現できる.しかし BWIDTH を 3 にして,3 ビットの配列がこのままでうまくいくか,というとそうでもなく,本来の目的の「ケチくさい」ことからは逸脱してしまう.また BWIDTH は CELLWIDTH を越えた値だと全然まずい.これらの問題にどう対処するかというのはプログラマ任せだが,やはりプログラミングというのは

特殊性と汎用性のせめぎあい

だと思えてくる.その証拠に,1 ビットの配列を実現したいのなら,今の方法よりずっとコード効率のいい方法が何通りもある.1 ビットの配列は,char の配列の 8 分の 1 で済むというのが画期的である.たとえば,エラトステネスのふるいなら,char の配列を使ったときの 8 倍の数まで素数探索できるし,グラフィック RAM に点を打ったり消したりするのにも利用できる(16 色なら 4 ビット配列かな?).実に強力だ.

これらのマクロは特殊な配列のアクセスであるので,できればその配列の大きさ(1 ビット配列なら 8 分の 1 ぐらいとか)もマクロで自動的に計算したほうがいい.最後にある entries は,配列が char で計ったときどれだけの大きさになるかを算定する.ただの割算だと,うまくいかないのは明らかだろう.

リスト 3-5 のようにシンボル化された式を見ると,埋め込みシンボル定数「_MB_」をパラメータにすることで,もっと汎用的に使えそうなのに気がつく.たとえば

```
#define rooms(n,cap) (((n) + ((cap) - 1)) / (cap))
```

としておけば，*n* 人のお客さんを *cap* 人部屋に「寿司づめ」るには，何部屋必要かというのを計算するのにも利用できるだろう(そんなのあるかあ？)．

デバッグ時にもマクロを使う

リスト3-5のマクロはデバッグメッセージの出力に利用できるマクロだ．

```
#ifdef DEBUG
# define dprintf(f,a1,a2,a3) \
                fprintf(stderr,f,a1,a2,a3)
#else
# define dprintf(f,a1,a2,a3)     /* nothing */
#endif
```

リスト3-5　デバッグ用の便利なマクロ

もしもプログラムにDEBUGというシンボルが定義されていれば，dprintfはstderrにメッセージを出力するが，DEBUGシンボルが定義されていなければ何もしない．このdprintfを使っておけば，デバッグが終わったときに，今までデバッグメッセージを出力するのに使っていたコードを，あわてて削除して回らずに済む．

プログラムの中で，デバッグメッセージを出力する部分は，DEBUGというシンボルを定義しておかなければ，消えてなくなる．

```
n = load_file(f_name);
/* debug message */
dprintf("%s(%d)lines\n", f_name, n, 0);
```

複数の文を定義したい

今度は，マクロの中で複数の文を実行したいときを考えてみよう．実際のプログラミングでは，結構こういう場合があるはずだ．たとえば，リスト3-6のように，文字列と改行コードを書き出したいようなケースである．

```
#define PUTLINE(s) \
{ \
        write(1, s, strlen(s)); \
        write(1, "\r\n", sizoef("\r\n")); \
}
```

リスト3-6　マクロ中で複数の文を定義

普通，マクロは関数呼び出しみたいなイメージで使うから，プログラム中では次のように書くはずだ．

```
PUTLINE("hello");
```

この場合，最後に

```
};
```

というプログラムテキストが生成されるが，〝}〟と〝;〟の間には空文が入っているとみなせば，文法的には何の問題もない．

しかしこのマクロを以下のような文脈で使った場合を考えてみよう.

```
if (**argv != '-')
        PUTLINE(*argv);
else
        PUTLINE("** This is option **");
```

するとプリプロセッサは，この部分を次のように展開してしまう.

```
if (**argv != '-') {
        write(...
};
else {
        write(...
};
```

つまり，elseの前に〝;〟があるからif文は終結していて，〝elseに対応するifがない〟という文法エラーが起こってしまうのである．これでは困る.

単純な解決策としては，次のようにif文の中でPUTLINEを使うときは必ず{}でくくるようにするという方法があるだろう.

```
if (**argv != '-') {
        PUTLINE(*argv);
} else ....
```

しかし，これはちょっと面倒くさいといえば面倒くさいし，1行だけなのに{}でくくるのは自分の方針に合わない人もいるはずだ．そこで，これが苦痛なあなたのために，別の解決手段を紹介しよう.

```
#define PUTLINE(s) \
if (1) { \
        write(1, s, strlen(s)); \
        write(1, "\r\n", sizeof("\r\n")); \
} else
```

ところが，これでもまだ問題がある．この定義のやり方だと，

```
PUTLINE("Hello"), PUTLINE("world !");
```

と書かれたときに対応できないのだ．しかしこんな風に文を〝,〟で続けて書くのが好きな人は，最初から

```
#define PUTLINE(s)     ( write(1,(s),strlen(s)), \
                  write(1,"\r\n",sizeof("\r\n")) )
```

と定義しておけばよい(これが一番いいかもしれない)．

複数の式を連続して評価させる演算子である〝,〟は，関数呼び出しのパラメータを並べる〝,〟と同じ文字であるが，意味は違う．〝,〟演算子は複数の式を〝左から〟評価し，最後の式の値を返すものである．a が int で 2 のとき

```
a += 1, a += 2
```

は 5 という値を持つ式になる．

basedポインタをマクロで実現しちゃう

MS-C ver.6.00では，basedポインタという新たなポインタクラスが採用された．今までのnear, farポインタの間に位置するようなポインタである．実際のマクロを紹介する前に，このポインタについて少し説明しておこう．

8086のアーキテクチャでは，セグメントとして64Kバイト単位のメモリブロックを割り当て，それをセグメントレジスタという16ビット値で識別し，セグメントの内部をアドレスするにはさらに16ビットの値のオフセットを用いている．もしも参照するセグメントが1つなら，オフセットだけを使えば済む．これをnearポインタと呼んでいる．そしてセグメントレジスタとオフセットのペアである32ビット値が，farポインタである．

farポインタの表現力(そのアドレス可能な空間の広さ)はnearポインタの比ではないのだが，如何せんnearポインタの倍の大きさを持ってしまうところが問題になる．そこで，なんとかならないものかと考え出されたのがbasedポインタなのだ．

basedポインタとして定義されたポインタは，nearポインタと同じ大きさ(16ビット)で，その基点となるベースポインタ(farポインタ)からのオフセットを持つ．実際のbasedポインタは次のように定義される．

```
int _far *a;
int _based(a) *p;
```

こうすれば，pは16ビットの大きさを持つ変数で，aを基点とするオフセットを持つようになる．つまり，aの方を変更してやれば従来のfarポインタと同様のアドレス表現力が得られるし，それと同時にnearポインタのメリットも活かせるのである．

では，MS-C ver.6.00で導入されたこのbasedポインタの機能を，本書ではマクロで実現してみよう．1つのやり方は，リスト3-7のようなマクロによるものである．

```
#define BPTR(type, base, p)      \
        ( (type far *)((char far *)base \
                        + (unsigned)p) )
```

リスト3-7　basedポインタを実現するマクロ(その1)

このときそれぞれのポインタは，

```
int far *a;
int near *p;     /* unsigned p; */
```

と定義しておく．そして，

```
*p = *p + 1;
p[5] = 123;
```

とアクセスするときは，以下のようにすればよいのだ．

```
*BPTR(int, a, p) = *BPTR(int, a, p) + 1;
BPTR(int, a, p)[5] = 123;
```

また，別のやり方も考えられるだろう．リスト3-8の例は，ポインタの宣言にもマクロを定義して，ポインタの参照を行うときにその型を指定しなくても済むようにするものだ．

```
#define DECLARE_BASED_PTR(type, p) \
        typedef type far *_type_of_##p; \
        type near *p;

#define BPTR(base, p) \
        ((_type_of_##p)((char far *)base \
                                + (unsigned)p))
```

リスト3-8　basedポインタを実現するマクロ(その2)

このやり方に従ってポインタを定義する場合は，

```
int far *bp;
DECLARE_BASED_PTR(int, p);
```

としておき，アクセスするときは，

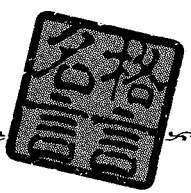
```
*BPTR(bp, p) = *BPTR(bp, p) + 1;
BPTR(bp, p)[5] = 123;
```

とすればよい．しかしこの方式だと，変数名と型が結び付いて固定してしまうので，同じ名前の変数を別の型で宣言できなくなる．もちろん型宣言のスコープはファイルの内部だけなので，ファイルを別々にすれば問題ない．

さて，実際にこのように定義された疑似basedポインタが，はたして本家basedポインタと同じように振舞うかどうかをチェックしてみよう．

リスト3-9のアセンブルリストを見てほしい．これは，MS-C ver.6.00で最適化を無効にして生成したものである(最適化を行っても同様の結果を得られるが，最適化しないほうが分かりやすい)．

アセンブルリストを見れば一目瞭然だが，疑似basedポインタが本家basedポインタと同じ振舞いを示すどころか，皮肉と言うべきか驚異と言うべきか，擬似basedポインタ版の方がよいコードが生成されている(!)．ちなみに，型を隠すマクロによる疑似basedポインタを使った場合も，これと同じコードが生成される．



バグの噂も七十五日

＜解説＞

自分のプログラムにバグが出たとき，75日間じっと我慢していれば，誰かが直してくれる（あるいはどんなバグだったか皆も忘れてしまう)という願い．逆の意味では，バグが出てから75日も放っておけば，そのプログラムは誰も使わなくなってしまい，自分はお払い箱になってしまうということ．

(A)MS-C ver.6.00 の本家 based ポインタ版

```
;
;
;  int far *a;
;  int _based(a) *p;
;
;  *p = *p + 1;
        les     bx,DWORD PTR [bp-6]             ;a
        mov     si,WORD PTR [bp-2]              ;p
        inc     WORD PTR es:[bx][si]
;  p[5] = 123;
        mov     bx,WORD PTR [bp-2]              ;p
        add     bx,10
        les     si,DWORD PTR [bp-6]             ;a
        mov     WORD PTR es:[bx][si],123
```

(B)マクロによる擬似 based ポインタ版

```
;
;  int far *a;
;  int near *p;
;
;  *BPTR(int, a, p) = *BPTR(int, a, p) + 1;
        les     bx,DWORD PTR [bp-6]             ;a
        mov     si,WORD PTR [bp-2]              ;p
        inc     WORD PTR es:[bx][si]
;  BPTR(int, a, p)[5] = 123;
        mov     bx,WORD PTR [bp-2]              ;p
        les     si,DWORD PTR [bp-6]             ;a
        mov     WORD PTR es:[bx+10][si],123
```

リスト 3-9　based ポインタの本家版と擬似版の生成コードの比較

トークンの合成

先の_based ポインタのところでも登場したが，2つ以上のトークンを合成し，あらたに1つのトークンとする機能が，MS-C の ver.5.10 以降や ANSI 準拠の C にある．それは〝##〟演算子によって行われ，先の例で

```
DECLARE_BASED_PTR(unsigned, pointer);
```

と宣言すると，

```
typedef unsigned far *_type_of_pointer;
unsigned near *pointer;
```

と展開される．この機能を，トークン結合演算子をサポートしていない処理系で実現するためのマクロが TOKEN と，_TOKEN である．このマクロを使えば，

```
#define TOKEN(a,b)        _TOKEN(a)b
#define _TOKEN(a)         a

#define DECLARE_BASED_PTR(type, p) \
        typedef type far *TOKEN(_type_of_,p); \
        type near *p;
```

と定義できる．

レコードを分解する

さて，マクロは，オブジェクトやその名前，型などを一元管理するのに，大変な威力を発揮する．といっても，なかなかピンとこないかも知れない．コンスタントデータをマクロによって一元管理することは，誰でも思いつくし簡単なことだ．

```
#define MAX_MEMBERS     123
```

と定義しておけば，MAX_MEMBERS さえ変更すれば，プログラムの中でその値を利用している部分は，すべて変更される．もしも，マクロを使用せず，プログラムの内部に〝123〟を埋め込んでしまうと，それらが同一の値であることを保証するのが面倒だし，変更も容易ではない．より複雑な場合として，関数のアドレスを含んだテーブルを構成することを考えてみよう．ここでは，LISP の処理系を作るような場合を例にあげるが，関数アドレスのテーブルを作るような一般の場合と同様だ．

LISP の各関数には名前と型，引数の数がある．C のプログラムで実現する場合には，それに加えて LISP の関数それぞれに対応した C の関数が存在することになる．これらを管理するためには構造体を使う．そして構造体の初期化と，C の関数(型を Object としておく)の宣言と，LISP での名前の定義をそれぞれ行う必要がある．

こうした宣言や定義は，たとえば関数名 CAR，型 SUBR，引数が 1 つという LISP の関数を実現したい場合には次のようになるはずだ．

```
/* Cの関数宣言とLISP関数の名前の定義 */
Object car(); char car_name[] = "CAR";  /* CAR */
...                          /* その他の関数が続く */

/* 関数テーブルの定義 */
struct nametab {
        Object (*func)();
        char *name;
        int type, argc;
} functab[] = {
        { car, car_name, SUBR, 1 },     /* CAR */
        ...                  /* その他の関数が続く */
};
```

しかし，本当に処理系を作ろうと思ったら，こうした関数ごとの定義を何百行も書いていかなければならない．骨が折れるのももちろんだが，上の方での宣言や定義とfunctabの中身の記述が食い違ってしまう可能性だって高くなる．関数はどこか1箇所でまとめてレコードのような形で属性を記述しておき，それを上記のようなCプログラムの形式に展開できたら，こんな楽なことはないはずだ．

それを実現する方法をここで紹介しよう．まず，関数ごとの属性(Cの関数名，LISPの関数名，型，引数の数)を，リスト3-10に示したfunc.hのように記述しておく．

```
/*
 * func.h
 */
FUNC(car, "CAR", SUBR, 1)
FUNC(cdr, "CDR", SUBR, 1)
FUNC(cons, "CONS", subr, 2)
FUNC(add, "+", LSUBR, 2)
FUNC(sub, "-", LSUBR, 1)
```

リスト 3-10　func.h

次に，C のプログラムは，リスト 3-11 のように記述する．

```
#define FUNC(symbol, name, type, argc) \
    Object symbol(); char symbol##_name[] = name;
#include "func.h"
#undef FUNC

struct nametab {
        Object (*func)();
        char *name;
        int type, argc;
} functab[] = {
#define FUNC(symbol, name, type, argc) \
        { symbol, symbol##_name, type, argc },
#include "func.h"
#undef FUNC
};
```

リスト 3-11　プリプロセス前のリスト

そして，リスト3-11のリストをプリプロセッサにかけると，リスト3-12に示したように，目的の宣言や定義がきちんと生成されるのである．

```
Object car(); char car_name[] = "CAR";
Object cdr(); char cdr_name[] = "CDR";
Object cons(); char cons_name[] = "CONS";
Object add(); char add_name[] = "+";
Object sub(); char sub_name[] = "-";

struct nametab {
        Object (*func)();
        char *name;
        int type, argc;
} functab[] = {
        { car, car_name, SUBR, 1 },
        { cdr, cdr_name, SUBR, 1 },
        { cons, cons_name, subr, 2 },
        { add, add_name, LSUBR, 2 },
        { sub, sub_name, LSUBR, 1 },
};
```

リスト3-12　プリプロセス後のプログラム

そもそもの目的はfunctabの中身を作ることだが，前述のように，そのためにはCの関数の宣言とLISP関数名の宣言が行われていなければならない．だからといって，それをまったく別々に記述するのではなく，同じfunc.hを元に一括して生成してしまうという点が，ここで紹介した技の賢いところなのだ．

この世でもっとも汎用的なCプログラム

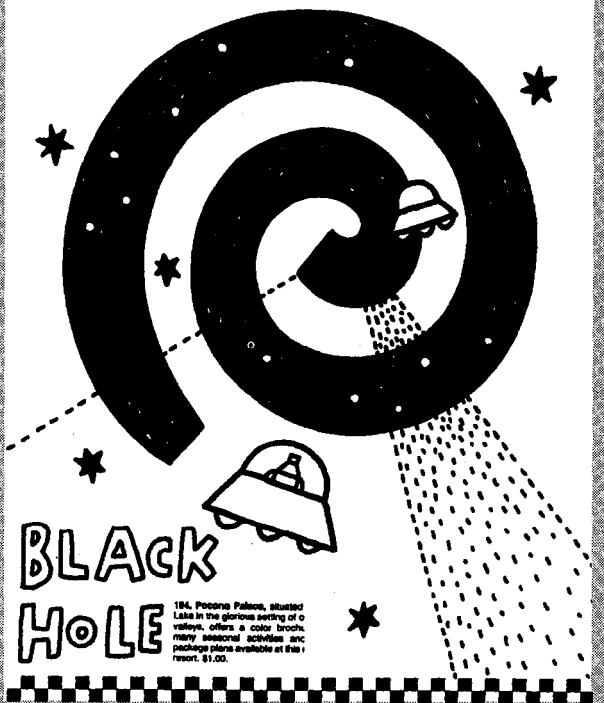
```
#include <con>                    /* include "/dev/tty" */
```

金の切れ目が¥の切れ目

＜解説＞

長いマクロを書くと，ハードウェア（金物）の都合上，一行に収りきらない場合がある．そこで，しかたなくバックスラッシュ(¥)で，マクロを複数行に分断して記述すること．この本のリストは典型的な例である．

標準コーディングテクニック



前の章ではプリプロセッサ(特にマクロ)の技を紹介した．Cを使いこなすには，もちろんそのほかにもいろいろな知識や技を身に付けておかなければならない．そのためには，大雑把に言って，C言語自体の理解を深めることと，Cのライブラリの知識を仕入れる必要があるのだ．ライブラリについては次章にゆずることにして，この章ではC言語プロパーの話題（構文やポインタ，データ型や構造体など）を取り上げることにしよう．

switch文をきちんと使う

まず手始めに，ごく初歩的な事例をあげてみよう．複数ケースからの選択処理において，switch 文を使わない人がいる．たとえば，次のように記述するプログラマだ．

```
if (*p == 'a') {
        なんたら
} else if (*p == 'b') {
        かんたら
} else if (*p == 'c') {
        どうした
} else {
        こうした
}
```

BASIC や FORTRAN だと C の switch 文のような制御構造がないので，こう書くのも仕方がないだろう．しかし，C には switch 文があるのだから，自分の過去に執着せず，使うべきところでは使って，他人が読んでも分りやすいプログラムを書くように心がけるべきだろう．また，最初は else if が 1 つくらいで済んでいたのが，プログラムの仕様の変更などで判定の条件が増えてしまったような場合にも，無意識にやたらと else if を増やすようなことはしないで，一気に switch で書き換えてしまった方がよい．

```
switch (*p) {
case 'a':
        なんたら
        break;
case 'b':
        かんたら
        break;
case 'c':
        どうした
        break;
default:
        こうした
}
```

最近はQuick BASICなんかにもSWITCH文があるぐらいだし，古い習慣にとらわれていると，まわりのプログラマから迫害されるのが落ちなので，気を付けるようにしたいものだ．

ところが，中には，

分って，やっている

人もいる．switch文は，都合がいい場合(MS-Cの場合，ラベルの値が等差数列で，その数が7個を越えるとき)には，比較をテーブルジャンプに置き換えるので高速なのだが，逆にテーブルジャンプにならない場合も多い．そのとき，switch文はどの順番で比較を行うのか保証していないので，無駄な比較が何度も行われる可能性がある．これを見越して，もっとも成立しやすい条件からif文で「ひっかけて」いるのかもしれない．もしも，コンパイラの最適化を見越したコーディングだとしたら，迫害すると，かえってこちらが笑われる羽目になる．

gotoは邪悪?

ところで, switch文がループの中にあるとき, switch文で判定した条件によっては, ループを脱出したいというケースもよくある. たとえば, リスト4-1のようなプログラムの場合だ.

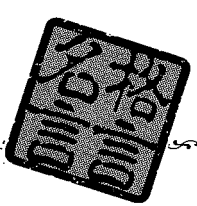
```
while ((c = getchar()) != EOF) {
        switch (c) {
        case '\t':
                なんたら
                break;
        case '\r':
                かんたら
                break;
        case '\n':
                どうした
                goto done;
        default:
                こうした
        }
}
done:
        あれこれ
```

リスト4-1　goto文を使って脱出

switch 文の中では，break 文を使ってループを脱出するわけにはいかない．こういう場合には，goto が嫌だからといって switch 文をわざわざ else if の嵐に直すより，リスト 4-1 で示したように，素直に goto を使って書くべきだろう．

「goto を使わないこと == 構造化プログラミング」ではない

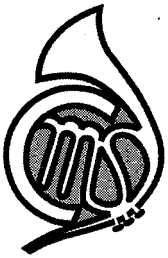
のだから．



それにつけても金の欲しさよ

＜解説＞

いくら高速化チューンを施してもハードが遅ければどうしようもないこと．あるいは主記憶が足りないばかりに，複雑なデータ構造を用いることを迫られたことを嘆く様子．強力なハードウェアを揃えるにはお金が必要であることと掛け言葉になっている．



関数まわりのコーディング

とまあ，まずは軽いジャブのような話をしたところで，次は関数がらみのコーディングの話題を取り上げよう．プロトタイプ宣言や引数に関して，いくつか気を付けておきたい問題があるのだ．

プロトタイプ宣言の落し穴

ANSI 規格で取り入れられた便利な機能として，プロトタイプ宣言がある．これは，関数の参照を宣言するときに引数の型まで記述しておくことにより，実際の呼び出しでの型の不一致をチェックできるようにする機能だ．

MS-C の標準ライブラリは，すべて適切なヘッダファイル中でプロトタイプ宣言されている．たとえば，fseek() は，stdio.h で次のように宣言されている(MS-C ver.5.10 の場合)．

```
int _CDECL fseek(FILE *, long, int);
```

最初の引数がファイルポインタ，2 番目がシーク開始位置からのオフセットを示す long 型整数，3 番目がシーク開始位置を示す定数である．したがって，たとえば次の文は，現在のファイルポインタの位置から 10 バイト後(ファイルの終端方向)の位置へシークするという意味になる．

```
fseek(fp, 10L, SEEK_CUR);
```

ところがこれを次のように書くと，2番目の引数がlong型でないので，間違った動作をしてしまう．

```
fseek(fp, 10, SEEK_CUR);
```

しかし，stdio.hをインクールドしてプロトタイプ宣言を行っておくと，この2番目の引数を自動的に型変換してlong型にしてくれる．MS-C ver.5.10では型が違うというワーニングが出るものの，正しく動作する．もちろん，ワーニングがうっとうしいなら，記述を正しく修正すれば済むだけの話だ．プロトタイプ宣言がないと，こういう間違いを発見するのはかなり大変になる．

このようにプロトタイプ宣言は非常に便利なので，

自分で書いた関数についても プロトタイプ宣言を積極的に使う

べきだ．特に複数モジュールからなるプログラムでは，各ソースに共通にインクルードされるヘッダファイルを作成して，その中にこのプロトタイプ宣言を記述しておくとよい．

しかし，逆にいえば，プロトタイプ宣言があると必ずこの自動型変換が行われるため，プロトタイプ宣言が間違っていたりすると，かえって誤動作の原因になってしまう．MS-C ver.5.10ではワーニングが出るので注意していれば問題ないといえば問題ないが，MS-C ver.6.00ではワーニングを出してくれないので，特に注意が必要になる．

間違ったプロトタイプを書かないためには，よい方法がある．コンパイラ自身にプロトタイプ宣言を作成させるのだ．MS-Cには−Zgというオプションスイッチがある．これを使って，次のようにコンパイルすると，ソースファイルfoo.c内で定義されているすべての関数のプロトタイプが画面に出力される．

```
cl -Zg foo.c
```

この出力を適当なファイルにリダイレクトしてセーブし，エディタでソースの先頭やヘッダファイルなどに切り貼りすればよいのである．

また，このときは，−Zs オプションも一緒に使うと便利だろう．このオプションは，オブジェクトを作らずにシンタックスチェックだけを行ってくれる．

引数として副作用のある式を書く

さて，プロトタイプ宣言を行ったからといって，これで引数は安心だなどと思うのは256年早い．

次のような関数呼び出しを考えてみてほしい．

```
a = 1;
sub(a, ++a, ++a);
```

このとき関数 sub は，引数としてどういう値を受け取ると予想されるだろうか．これは，実は，

分らない

というのが正解だ．3つの引数が左から右の順に評価されるなら

```
sub(1, 2, 3);
```

となるだろうし，右から左に評価されるなら

```
sub(3, 3, 2);
```

となる．実はCの言語仕様では，関数呼び出しの際の引数の評価順序は，

処理系に依存する

とされているのだ．したがって，これはどっちになるのか一様に保証されはしない．

現実には，おそらくスタックにPUSHされる順番に評価されるのだろう．だとすると，MS-Cではpascal宣言した関数は前者，cdecl宣言の関数は後者になると思われる．しかし，仕様として保証されているわけではないので，これに依存したプログラムを書くのは危険だ．

先に上げた例からも明らかだが，関数の引数の中に副作用のある呼び出しは書くべきではない．もちろん，実際に書いてみればそれなりに動くプログラムにはなるだろうが，こういうのを書いてると，まず間違いなく，

プログラミング能力を疑われる

羽目になるので，絶対にやめるようにしたい．

引数の渡し方

さて，引数関係の話題はまだある．

関数で処理するための変数や値を渡すには，引数として渡す方法と，共通な外部変数を介して渡す方法がある．Cの場合，引数は必ず値渡しになる．つまり，変数を引数として与えると，その値をコピーして渡すので，元の変数の値を変化させずに済むわけだ．逆に，渡した変数の値を変化させたい場合は，そのアドレスを渡して，関数の方はポインタで受けることで処理する．一方，いくつかの関数で同じ変数を操作する場合には，いちいち引数にするよりは外部変数にした方が明解に書けるケースもある．

まあ，こんなことはCをきちんと理解している人なら今さら説明される

までもないことだろう．しかし，具体的に何を引数として，何を外部変数とするべきか，という問題は結構難しい．

何でもかんでもポインタ渡しにするやつ
(FORTRAN 出身者に多い?)

もいれば，

何でもかんでも外部変数のやつ
(BASIC 出身者に多い?)

もいる．こういうのはほんとに人それぞれで，その人なりのルールのようなものを持っていれば，どうでもよいことだといえなくもない．

だが，一般論としていえば，その関数だけで使うものなどは引数にした方がよい．外部変数にはどこで参照されているか分らないというデメリットがあるからだ．引数にした方がよいものまでむやみやたらと外部変数にすると，プログラムの見通しが悪くなってしまうのだ．

main()で引数を受け取る賢いやり方

一方，関数側では渡された引数を適切に処理する必要がある．ここではその1つの特殊なケースである、main()での引数の処理方法について述べよう．

Cで書かれたプログラムには，起動時にパラメータを渡せる．このパラメータは，main()への2つの引数として与えられ，そのうち1つ(argc)はパラメータの数(プログラムが起動されたときの名前も含む)，そしてもう1つ(argv)がパラメータ文字列の配列(正確には，パラメータ文字列へのポインタの配列のベースを指すポインタ)である．

初心者は，この引数の扱いがわからない(パラメータの意義すら理解しよ

うとしない)ので，プログラム中のfgets()によって，ユーザーにパラメータ文字列の入力を強要する．プログラムの起動から，パラメータの指定の間に(ロードの)タイムラグがあることは，キーボードに慣れた人間にとっては苦痛であるし，何よりも，fgets()をパラメータ入力に使うということは，標準入力をつぶすことにもなる．パラメータをコマンドラインで指定しきれないようなプログラムは別にしても，せっかくコマンドラインパラメータを「取って」これるのだから，利用しない手はない．

しかし，argcとargvの扱いは慣れないと(慣れても？)面倒だというのも事実である．なにが面倒なのかというと，argvをアクセスするときに，ちゃんとパラメータの数をargcを使ってチェックしなければならないからである．バイブルはargcの数だけargvの配列要素が存在することは保証しているが，それ以上はなにも言及していないので，コマンドラインパラメータを出力する場合は，リスト4-2のようにしなければならないのである(argvの指す配列の先頭にある文字列は，プログラムの起動されたときの名前である．またANSIの標準規格案ではargv[argc] == NULLが保証される)．

```
#include <stdio.h>
main(argc, argv)
char **argv;     /* int argc; の宣言は省略可能 */
{
        while (--argc > 0) {
                argv++;
                puts(*argv);
        }
        exit(0);
}
```

リスト4-2　コマンドラインパラメータを出力する

ややこしくなるのは，プログラムがオプションとファイル名のように，性質の違うパラメータを持つときである．一般に，C で書かれたプログラム(あるいは UNIX 文化のプログラム)は，つぎのようなコマンドラインで起動される．

```
program -<opt> ... <file> ...
```

<*file*>に与えられるのが，処理の対象となるファイルで，ハイフン〝−〟に続いて与えられる<*opt*>が，処理のオプションスイッチである．さらに具体例を示すと，

```
sort -n -r score.1 score.2
```

は〝score.1〟と〝score.2〟をまとめて，各行の数値に従って，逆順に並べ換える．

このコマンドラインルールに従い，main()の引数を正しく処理しようとすると，リスト 4-3 のように書ける．

```
#include <stdio.h>
main(argc, argv)
char **argv;
{
        while (--argc > 0 && **++argv == '-')
                parse_opt(*argv + 1);
        while (argc-- > 0) {
                process_file(*argv);
                argv++;
        }
        exit(0);
}
```

リスト 4-3　引数の処理

初めのオプションを解釈するループと，後のファイル名を解釈するループで，argc のデクリメント，argv のインクリメントのタイミングが全然違うところが，

なんともミラクル

なのである．

今紹介したような数行のコードであれば，頭で追っていくのにさほど苦労はいらないが，このループの中でさらに，argc や argv を操作してしまうと，だんだん分らなくなってきて，突然，正しく動かなくなってしまうこともある(もちろん正しくないコードを書いたのは自分である)．ただでさえ(argc と argv の)二元管理なのに，それを変更するコードが，あちこちに散らばってしまうのが問題なのだ．

情は人の為ならず

＜解説＞

人の書いているプログラムに文句（アドバイス）をつけるのは，後々メンテナンスすることになるかもしれない自分のためだということ．あるいは単に「やじ」をとばして面白がっているだけのこと．いずれにせよ，プログラムを書いている本人のためではない．

1つの美しい解決策はリスト4-4のようにint indexを用意し，argvをポインタの配列であるとみなして，indexによってアクセスすることである(argcは配列の添字の上限とみなす)．

```
main(argc, argv)
char *argv[];
{
        int index;

        for (index = 1; index < argc; index++)
                if (*argv[index] == '-')
                        parse_opt(argv[index] + 1);
                else
                        break;
        for ( ; index < argc; index++)
                process_file(argv[index]);
        exit(0);
}
```

リスト4-4　indexを使って引数を処理する

この方法が「洗練」されているかどうかは個人の主観にもよるが，初心者にも分かりやすいコーディング例だと言えるだろう．しかしこれでは

欲求不満

だという人のために，もう1つコーディング例(リスト4-5)を示そう．この例はループが1つで，argcとargvの変更箇所も散らばらず(1箇所)，ある特殊なオプション構文規則と，デフォルトの標準入力をサポートする．マクロ〝process_file〟と〝load_script〟は実使用する場合は関数になる(ここでは，問題の本質ではないので，ただのprintf())．

```
#include <stdio.h>

char stream_sym[] = "-";

#define at_opthead(p)        \
            ((p)[0] == '-' && (p)[1] != '\0')
#define process_file(str)    \
            printf("processing file : %s\n", str)
#define load_script(str)     \
            printf("loading script : %s\n", str)

#define TRUE                 (1 == 1)
#define FALSE                (!TRUE)
#define SCRIPT_IS_REQUIRED   TRUE

parse_opt(s)
char *s;
{
    while (*s != '\0') {
        switch (*s) {
        case 'f':
            if (*(s + 1) == '\0')
                return SCRIPT_IS_REQUIRED;
            else {
                load_script(s + 1);
                return FALSE;
            }
            /*NOTREACHED*/
        case 'n':
            printf("'n' option is accepted\n");
            s++;
            break;
        default:
```

```
            fprintf(stderr,
                "%c : unknown option\n", *s);
            exit(1);
            /*NOTREACHED*/
        }
    }
}

main(argc, argv)
char **argv;
{
    int in_option, script_file;

    script_file = FALSE;
    in_option = TRUE;
    while (--argc > 0) {
        argv++;
        if (script_file) {
            load_script(*argv);
            script_file = FALSE;
        } else if (in_option && at_opthead(*argv))
            script_file = ( parse_opt(*argv + 1)
                        == SCRIPT_IS_REQUIRED );
        else {
            in_option = FALSE;
            process_file(*argv);
        }
    }
    if (in_option)
        process_file(stream_sym);
    exit(0);
}
```

リスト4-5　引数処理(洗練版)

MS-C ver.6.00の_fastcall

ところで，MS-C ver.6.00には引数をレジスタ経由で渡す機能がある．これは，pascal宣言とほぼ同じだが，最大3つまでの引数をスタックではなくレジスタを使って渡すものだ．

```
void _fastcall sub(short a, short b, short c);
```

このように宣言した関数では，a，b，cはそれぞれAX，DX，BXレジスタを介して渡される．long型の場合は，DX：AXレジスタペアを使って渡せる．

```
void _fastcall sub(long d, short c);
```

こう記述すると，dがDX：AX，cがBXレジスタを介して渡されるようになる．しかし，これを

```
void _fastcall sub(short c, long d);
```

と記述すると，cがAXとなって，dはスタック経由で渡されることになってしまう．引数の順番に注意しないと，効果が上がらないのである．

引数をレジスタ渡しにすると，スタックにPUSHする手間がなくなる分だけ処理が高速になるが，現実には複雑な関数を_fastcall指定しても，それほど効果がない．なぜかというと，複雑な関数ではレジスタの使い回しがうまくいかないので，受け取った引数を結局一度スタックにPUSHしてから，AX，BXなどのレジスタを使うというコードが出力されるケースが多いからなのだ．この辺は，ぜひアセンブラソースを出力して，じっくり研究してみてほしい．

データ型に潜む罠

データ型というのは，プログラミングの初心者とか，BASICみたいな変な言語しか使ったことのないプログラマには，何のためにあるのかなかなか理解できないかも知れない．しかし，Cを何年も使っているプログラマでも，データ型をうまく使いこなせない人もいる．うまく使いこなすというのは，別に「型を厳密に区別する」とか「ちゃんとキャストする」というようなことだけではないのだ．この節を読んで，データ型に潜む意外な罠を理解するようにしてほしい．

オーバーフロー!!

BASICなんかと違ってCには色々なデータ型があるが，基本の型は次のものだけである．

```
char    int     float    double
```

こうしたデータ型の大きさのうち，charはターゲットマシンの文字コードを表現するのに十分な大きさのデータ型で，intはターゲットマシンにとって最も自然なデータ型であるとされている．このことはバイブルに書かれているとおりだ．

そして，この型にshortやlongの限定詞を付けたり，それを指すポインタとして型を定義できる．これらのオブジェクトの，主記憶中に占める領域，つまりデータオブジェクトのサイズを決めつけることはできない．つまり，charは8ビットとは限らないし，intは32ビットとも限らないので

ある．これにshortやlongの限定詞が付くと，問題はさらに複雑になる．shortはintより大きくなければよく，longはintより小さくなければよいのだ．最近ではこれに，shortとintは少なくとも16ビット，longは少なくとも32ビットという制限が加わったが，データオブジェクトのサイズを勝手に決めつけてはいけないことには変りがない．

世の中には，charは9ビットで，short，int，longとも36ビットというマシンも存在する．これは特殊な例だとしても，データオブジェクトのサイズを決めつけたプログラムは，移植できないプログラムになってしまう．

たとえばMS-Cでは，intは16ビットで，longは32ビットである．longの半分の大きさである16384がaとbに入っているとき，次のコードは正しく動作する．

```
int a, b, c, d;
d = (int)((long)a * b / c);
```

ところが，intもlongも32ビットであるようなマシンでは，掛け算の時点でオーバーフローし，間違った答えがdに代入される可能性が大である(つまりaとbにlongの半分の大きさである2^{30}が入っているとき)．当り前のことかもしれないが，実際にはこのようにコーディングしてしまっている場合は少なくないと思う．

さて，intのオーバーフローの話が出たところで，プログラマの心得を3つばかり紹介しよう．

まず第一は，

atoi()を気安く呼ぶな

ということだ．atoi()を気安く呼んではいけない．言い換えると，atoi()で返された値を軽々しく信用してはならないのである．なぜならば，atoi()はintがオーバーフローするという現実に背を向けているからだ．たとえば，

int が 16 ビットのマシンで，184184 という文字列を atoi() に渡すと，平気な顔をして −12424 を返してくる．したがって，重要な数値を入力するような場合には，atoi() を使ってはいけない．

2 番目の心得はこれだ．

掛け算には注意しろ

先の例とも重複するが，気軽に掛け算をするというのはとても恐ろしいことである．掛け算は，簡単にオーバーフローを引起こすからだ．正しいやり方は，掛け算の結果がデータオブジェクト (32 ビットとも限らない) に入り切ることをプログラマが保証してやるか，掛け算をしないかのどちらかである．

そして，

加減算にも注意しろ

というのが 3 番目だ．これまで書いてきたことからもう明らかだろうが，加減算にも注意しなくてはならない．オーバーフローの危険はここにもある．配列の大きさや，要素の数などを unsigned で数えるというのは言うまでもないことだが，unsigned にしたからといっても安心するのはまだ早い．16 ビット unsigned int で，65535 に 1 を加えると，0 になってしまうのは分りきったことだ．65535 に 1 を加えない，あるいはそんな状況を起こさないようにするのは，プログラマの責任なのだ．

今までの話は，重箱の隅をつつくような，偏執的な話ではない．プログラムのバグ，しかも見つけにくいものの大半は，この手のものなのだ．

charの型変換に注意しろ

さて，データ型の問題の中でも型変換にまつわる問題はただでさえいろいろと厄介なのに，日本語処理が絡んでくると話が一層ややこしくなる．

ここでは，話を簡単にするためにcharが8ビット，intが16ビットであると仮定しよう．charが8ビットである場合，最上位のビットが立っていればその値は負である．だが，文字コードとしてASCIIを使用している限り，すべての文字コードは128より小さいので，最上位ビットが立つことはない．つまりすべて正なのである．そのため，文字をただのcharで扱おうが，unsigned charで扱おうが変りはない．

しかし日本語のコードは，その値が128より小さいとは限らず，128より大きいコードを持つ文字も存在する．これを8ビットの(符号付き)charで扱おうとすると，当然ながら負の値となる．ここで問題が持ち上がるのだ．

```
*s == 0x81
```

この式を評価するとき，sがcharへのポインタだと正しく動作しないのである．この問題を少し詳しく説明しよう．

暗黙の型変換で，*sの値はintに拡張される．だがこのとき*sの型はcharなので，符号拡張が起こるのだ．だからもし*sの値に0x81相当のものがきても，＝＝の左辺として評価されるときには，その値が0xff81になってしまうのである．これに対し右辺の0x81はintの0x0081のことだから，そのままである．結局この式はプログラマの意図に合致しないし，真になることもない．だからといって，これを次のようにしても解決にはならない．

```
(unsigned)*s == 0x81
```

なぜなら，*sが符号拡張された後にunsigned intに型変換しても，拡張された16ビットの値(たとえば0xff81)は変わらないばかりでなく，負の値を正の値に変換するという，妙なことになる．

これを解決する1つの方法を示そう．

```
*s == (char)0x81
```

この方法だと，左辺の符号拡張と同様に右辺もintの0x0081が(符号付き)charに型変換され，それが暗黙の型変換によってintに符号拡張される．そして結局右辺は0xff81になるのだ．

もう1つの解決策を示す．

```
(unsigned char)*s == 0x81
```

このとき左辺に起こる変換を考えると，(符号付き)charが暗黙の型変換によってintに符号拡張され，それがunsigned charに型変換され，そしてもう1度，暗黙の型変換によってintに「無」符号拡張(ゼロ拡張)される，と考えられる．

そしてさらなる，より単純な解決策は

```
*(unsigned char *)s == 0x81
```

である．この場合，sがunsigned charへのポインタに型変換され，*演算子によってunsigned charの値が取り出され，それが暗黙の型変換によって「無」符号拡張される．つまり，最初からsをunsigned charへのポインタとして宣言しておけばいいのである．

では，実際にMS-Cでこの式がどのようにコンパイルされるかをリスト4-6に示そう(BXレジスタがポインタsに相当する)．

```
; signed char
    mov al,BYTE PTR [bx]
    cbw                              ; 符号拡張
    cmp ax,129

; unsigned char
    cmp BYTE PTR [bx],129            ; 暗黙の型変換を
                                     ; 省略している！！
```

リスト4-6　signed charとunsigned charの型変換の違い

MS-Cではデフォルトのcharをunsigned charとして扱うオプション−Jがあるが，このオプションを使うと，プログラムを見ただけだとその〝魔法〟が分らないので，よくない．コメントとして書いておくか，−Jを使わずに，unsigned charを使った方が，後々のためだろう．

ではここで，格言を一発．

日本語テキストを含んだ文字列を扱うとき，
文字の型はunsigned charとせよ!

究極の添字

さて，ここでがらりと話題を変えて，ポインタの話の前段階として，配列の添字について述べよう．

まだポインタに慣れていない人は，とかくデータを配列として扱って，添字を使いたがるものだ．逆に，ポインタが身体に馴染んでしまうと，添字を使うのが何となく照れくさい(?)感じがして，宣言以外では[]演算子なんてほとんど使わなくなり，何でもかんでもポインタでアクセスしたりする人もいる．ポインタと添字の使い分けは後述することにして，まずは添字の技から紹介することにしたい．

こういうコードを見たことがあるだろうか．

```
p[-2] = 'a';
```

[]という記号は演算子であって，演算子の優先順位の表を見ると，一番上に書いてあるはずだ．だから，この記号は配列だけでなくポインタにも使えるのである．演算子としての意味は，

```
p[i] == *(p + i)
```

ということだ．

したがって，ポインタを使って文字列を処理している場合に，たとえば2文字前の文字を得るために，

```
c = *(p - 2);
```

なんて記述をする場合があると思うが，これは次のように書いても同じことになる．

```
c = p[-2];
```

まあ，これが分かりやすいかどうかは別として，文法上は正しい．もちろん，アドレスｐ－２に有効なデータがあるのが確実でない状態でこういう記述をしないよう，プログラムで注意する必要があるのはいうまでもないだろう．

また，まれに次のようなコードも見受ける．

```
for (i = 0; i < 6; i++)
        put("string"[i]);
```

"string"[i]という記述は，あまり見たことがないと思う．実際に使用されるような例は，

```
for (i = 0; i < 2; i++)
        putchar("S M TuW ThF S "[wday * 2 + i]);
```

というような場合である．

先にも言ったように[]は演算子なので，これの前や[]の中に書けるのは，値を持つ「式」である．そして，〝string〟という文字列定数は，その先頭アドレスを値として持つ式なのだ．また，整数型変数ｉももちろん式なので，こういう記述が許されているのである．要するに，これは

```
static char p[] = "string";
```

のときのp[i]と同じ意味なのだ.

現実にはあまりこういう記述をするメリットはないだろう.文字列定数は,静的な配列として宣言して,使い回すというのが普通だからだ.

なお[]は,これの前の部分と括弧にはさまれた部分の2箇所に式を記述する演算子だが,それぞれの式の型は特に規定されているわけではない.だから,次のような記述も間違いではない.

```
c = i["string"];
```

一見,わけが分らないが,次のようにポインタに置き換えてみれば,明確になるだろう.

```
c = *(i + "string");
```

まあ,こういう記述はソースをわざと読みにくくすること以外の目的には,特に有用ではないかもしれないが,こういう風にも書けるのだということを知っておいても無駄にならないだろう.この特殊な配列アクセスが,本当に正しく動作するかという疑問もあるだろうが,我々がテストした処理系(MS-Cも含む)ではすべて正しく動作した.

添字とポインタの使い分け

今述べたように,次の式が真になることは仕様として保証されている.

```
p[i] == *(p + i)
```

しかし,この両者をきちんと使い分けると,周りから一目置かれるはずだ(知らないけど).同じ意味を表す次の2つの関数呼び出しを考えてみよう.

```
write(fd, buf, sizeof(buf));
write(fd, &buf[0], sizeof(buf));
```

この場合はbufを大きさsizeof(buf)のバッファとして「ひとまとめ」に扱っているので,&buf[0]という記述は分りにくくなるだけで無意味だろう.だから,上の呼び出しの方がよい.

一方,たとえば文字列の先頭2文字を調べるとき,次のように書くのは統一性がなくてよくない.

```
if (*p == 'A' && p[1] == ':')
```

これは,

```
if (*p == 'A' && *(p + 1) == ':')
```

あるいは

```
if (p[0] == 'A' && p[1] == ':')
```

のどちらかに統一した方がよいだろう.

まあなんだかんだいっても,中には山田君(仮名)のように,

配列として宣言したものは,
あくまでも配列としてアクセスすべきだ

という考え方をとるプログラマもいる.配列をポインタによって(あるいはその逆)扱えるというのはCの特徴の1つなのだが,これを柔軟性ととるか,いい加減と解釈するかはプログラマそれぞれの自由だろう.それでも配列とポインタが違うという根拠は,つぎのようなものである.

本来の配列のベースは，l-value を持たない単なる定数であり，ポインタは l-value を持つ「変数」である．簡単に言えば，ポインタには代入ができるが，配列のベースには代入はできない．引数をポインタであるとみなすマクロに配列を渡すと，エラーが発生する．この場合は簡単に対処できるが，ファイルの外の配列を extern でポインタとして宣言して，コンパイラでもリンカでも発見できない，凶悪エラーを作り出すこともできる．

次の例を見てもらいたい．ファイル f1.c の内容が，次のようになっているとしよう．

```
/*
 * f1.c
 */
int buf[] = { 11111, 22222 };
```

で，ファイル f2.c の内容が，

```
/*
 * f2.c
 */
#include <stdio.h>

pointer_access()
{
        extern int *buf;

        printf( "pointer access %d %d\n",
                *(buf + 1), *((int *)&buf + 1) );
}

main()
{
```

```
        extern int buf[];

        printf("array access %d\n", buf[1]);
        pointer_access();
        exit(0);
}
```

この2つをコンパイルしてリンクすると，

エラーはなにも発生しない

のである．しかし出力は次のようになってしまう．

```
array access 22222
pointer access -11839 22222
```

bufをポインタとして宣言して〝*buf〟によってアクセスすると，正しく動作しないのだ．実際にアセンブラコードを見ると，リスト4-7のようになっている．

```
; extern int *buf;
; printf( "pointer access %d %d\n",
;         *(buf + 1), *((int *)&buf + 1) );
;
    push    WORD PTR _buf+2    ; *((int *)&buf + 1)

    mov     bx,_buf            ; *(buf + 1)
    push    WORD PTR [bx+2]

    mov     ax,OFFSET DGROUP:$SG159
    push    ax
    call    _printf
```

```
; extern int buf[];
; printf("array access %d\n", buf[1]);
;
    push    WORD PTR _buf+2    ; buf[1]

    mov     ax,OFFSET DGROUP:$SG163
    push    ax
    call    _printf
```

リスト 4-7　アセンブラコードの中身

ポインタとして宣言した"buf"は，l-value(代入される値)を持っているので，その評価が余分な間接アドレッシングになってしまう．つぎの引数のところに，凶悪なキャストを使った式があって，この式は正しく"buf"をアクセスできる．つまり，むやみやたらにポインタと配列を混同して使うことは，このキャストと同じくらい"凶悪"だと考えることができないだろうか(もちろん今の意味が分っていて，便宜的に混同して使用している方々にまで，意見する気はまったくない)．

ポインタの完全制覇

先ほども少し触れたが，Cの初心者にとって，ポインタというのはどうもとっつきにくい存在のようだ．アメリカなどでも，配列にして添字ばかり使う人がけっこう多いという．ポインタをマスターするには，常にマシン上のメモリイメージを念頭に置きながらプログラミングする習慣を付けるとよいかも知れない．いずれにせよ，本書の一番最初でも述べたように，アセンブラというかマシンレベルの知識は欠かせないと思う．

だが，多少ポインタが使えるようになっても，それでポインタがマスターできたと思うのは256万年早い．ポインタを本当に駆使できるようになるには，なんといってもプログラミングやポーティングの経験や知識がものをいうのだ．この節を読んで，ポインタ使いの上級者へのステップとしてほしい．

ポインタはintではない!

ポインタはintと同じではない．というか，ポインタもintも処理系やマシンによってサイズがまちまちで，必ずしも同じサイズだということが保証されていないのだ．しかし現実には，BSDのフリーのソースを見ると，int型と同じとして扱っている例が多くある．こういう扱いは本来は間違いなのだが，これはBSDがインプリメントされたVAXというミニコンが，int型もポインタも32ビット値とすることができたので，混同しても実害がなかったためだ．

ところが，MS-Cではこういう扱いはできない．メモリメデルによってポインタは16ビットだったり32ビットだったりするのに，int型は常に

16ビットだからだ．ポインタをintと混同したようなコードを書くには，スモールモデルでなくてはできない．まあ，しかしやはりそういうコーディングは避けるべきだろう．きちんとキャストを使って記述しないと，わけの分らないプログラムになりかねないからだ．

それでもポインタはintだ!

(とはいうものの，ポインタをintと同一視できないのは邪悪だという信念を持っているプログラマも決して少なくない．この項では，そのひとり，山田君(仮名)の主張を紹介しよう．)

ポインタにnearポインタとfarポインタの2種類が存在することは，次の教訓を強く思い出させる．

ポインタはintではない!!

これは我等がバイブル(K&R)にも書かれていたことだ．バイブルが書かれたころのCは，PDP-11をターゲットとして書かれたものであって，ポインタとintは同じ大きさを持つものだった．そのため，これを同一視して使う輩がいた(実際intを使ってメモリをアクセスしても何の問題もなかったし)からこそ逆に，わざわざこんなことを言わなければならなかったのだ．いまでこそ，strcat()のようにポインタを返す関数は，その関数を呼ぶ前に，

```
char *strcat();
```

のように宣言するが，昔はそんな宣言はしなかった．宣言をしないとすべてintであると解釈される(strcatの戻り値もしかり)．そして，そのintを使って(ポインタに代入して)メモリアクセスしても，一向に構わない．ポ

インタもintでカタがつく(型がつく?)なら，なにも煩わしい型宣言のタイピングをしなくてもいいじゃないか．そもそもCらしさというのは，なるべく打鍵の数を減らすことにあるのだ．なんてピュアな世界だろう．

いまでもこの名残は残っているのかもしれない．プログラムでstrcat()を使うとしてもstrcat()がcharのポインタを返すなどとはわざわざ宣言しない．そのかわりstring.hのような型宣言をしてあるファイルをインクルードする習慣がついている．

MS-Cでもスモールモデルのnearポインタを使っていれば，ピュアな世界に住んでいられる．intもポインタもどちらも16ビット値だからだ．しかしラージモデルだと，ポインタは32ビットでintは16ビットのままだから，うまく動かない．

実際に痛感するのは，32ビットUNIXマシン用に開発された高度な(たとえば，光子魚雷を発射しながらワープ9で後退する)プログラムをMS-Cを使ってMS-DOSにポーティングしようとしたときだ．プログラムは巨大で，また巨大なデータを必要とする．だからMS-Cでコンパイルするときはラージモデルにしなければならない．ほとんどのUNIXマシンがポインタとintが同じ大きさなので，先に示したなピュアな書き方をしてある．ああもうこれ以上は話したくない．

ポインタとintを同一視することの是非について，読者も様々な意見があるだろうが，私[*1]は同一視できるものだと思う．intは配列のインデックスとして用いることができる．もしもこの配列が，主記憶の最小単位(バイト)の配列であり，その配列が主記憶のすべてをマップできる可能性があるなら(論理的には可能だ)，そのインデックスは最小単位へのポインタと同じように稠密(dense)だといえるからだ．実際，CPUレベルではintもアドレスも区別がない．CはPascalの代わりでなく，アセンブラの代わりのはずだ．

もしもポインタとintを同一視できないものなら，徹底的に区別して使用するべきだ．しかしそうなったとき，CはCでなくなってしまうような気がするのである．

---

*1 山田君(仮名)

far キーワードをガンガン使え

さて，本書では far ポインタと near ポインタについての話題がたびたび登場するが，MS-C でプログラミングする場合にはけっこう便利な機能なので，ここでも説明しておこう．

メモリモデルという概念は MS-C ver.3.00 の登場以前にもあったが，near，far といったキーワードを導入したのは MS-C が最初である．これらのキーワードを使ってデフォルトと違うサイズのポインタを扱えるようになったため，MS-DOS での C プログラミングの自由度がかなり向上した．うまく far ポインタを活用すると，スモールモデルでほとんどの仕事ができるようになる．

たとえば，PC-9801 のテキスト VRAM をクリアするコードを考えてみよう．far キーワードを使うと，次のように書ける．

```
unsigned far *p = (unsigned far *)0xA0000000L;

for (i = 0; i < 0x1000; ++i)
        p[i] = 0;
```

far キーワードがなかったら，スモールモデルやミディアムモデルでこういうコードは書けないことになる．

また，スモールモデルでプログラムを作成していると，コードやデータは 64K バイトまでしか使えない．しかし，これも far キーワードを使えばうまく対処できる．

たとえば，データの一部が入りきらないならば，あまり速度的に重要でないものを 64K バイトの外に追い出してしまう．エラーメッセージの配列なんかを追い出すと効果的だろう．

```
char far errmsg[][30] = {
        "out of memory.",
        "incorrect dos version.",
        ...
};
```

こういった感じでエラーメッセージなどをfar配列にしてしまえば，64Kバイトのデータセグメントから追い出せるのだ．これを固定長の配列にしている点に注意してほしい．もし次のようにポインタの配列にしてしまうと，文字列定数が64Kバイトのデータセグメント内に取られてしまう．

```
char far *errmsg[] = {
        "out of memory.",
        "incorrect dos version.",
        ...
};
```

もちろん，far配列だと，スモールモデルの標準ライブラリではほとんどサポートしていないので，いろいろと不便なことがある．実際，MS-C ver. 5.10ではprintf()の%Fs指定で文字列を表示することぐらいしかサポートがないのだ．MS-C ver.6.00になると，_fstrcpy()などといったfar文字列を操作する関数群が追加されているので，それでもいくらかましにはなっている．

さて，コードが64Kバイトのセグメント内に収まらない場合も，一部の関数をfar指定すればセグメントの外に追い出せる．その場合，次のように関数にfarを指定して定義すればよい．

```
int far subroutine(int a, char b)
{
        .
        .
        .
}
```

このとき，far指定した関数だけを集めたモジュールを作成し，それをコンパイルするときに−NTオプションを使ってコードセグメント名を指定する必要がある．−NTオプションを指定しないと，far指定した関数も結局デフォルトのコードセグメント_TEXTに取られてしまうので意味がないからだ．コードセグメント名は次のような感じで指定すればよい．

```
cl -c -AS -NT FARCODE farcode.c
```

ここではコードセグメント名をFARCODEと指定しているが，_TEXT以外なら別になんでもかまわないので適当な名前を考えればよい．

ところで，デフォルトのコードセグメントの外に置いた関数から，スモールモデルの標準ライブラリの関数は呼び出せない．標準ライブラリ関数はnearコールで呼ばれるように作られているからだ．だから，外に追い出す関数についてはこの点に気を付けて作成する必要がある．

上記のようなテクニックを使うと，スモールモデルでも複数のコードセグメントやデータセグメントを持つプログラムが書ける．できるプログラマは，セグメントを嫌がらずに積極的に活用するものなのだ．

絶対アドレスのコール

絶対アドレスをCプログラムからコールするには，どう書けばよいと思うだろうか．まあ，MS-DOS上でプログラムを記述している場合，あまりこういう要求が生じることはないが，まれにROM内ルーチンをコールしたりする必要が出てくるはずだ．

絶対アドレスをコールするには，far関数へのポインタを使えばよい．

```
void (far * func)();
func = (void (far *)())0xFFFF0000L;
```

こうすると，

```
(*func)();
```

でアドレス0xFFFF：0x0000をコールできる．いちいちポインタ変数を使うのが面倒なら，次のようにコールしてもよい．

```
(*(void (far *)())0xFFFF0000L)();
```

ちょっと分りにくい記述かも知れないが，キャストの意味をきちんと理解してから使ってほしい．これは実際に正しいコードを吐く(ただしVer.4.xまでの話で最新版での結果は皆様で試してみていただきたい)．

```
; (*(void (*)())0xffff0000L)();
    sub ax,ax
    mov dx,-1
    mov [bp-2],dx
    mov [bp-4],ax
    call    FAR PTR [bp-4]
```

一方，Turbo CにはMK_FP()というマクロがある．これは，

```
MK_FP(segment, offset)
```

といったように使って，セグメント値とオフセット値からfarポインタを作り出すマクロだ．MS-Cにはこういうマクロはないが，次のように簡単に作れるので，自分で用意しておくと便利だろう．

```
#define MK_FP(seg,off) \
    ((void far *)((unsigned long)(seg) << 16 \
            |((unsigned long)(off) & 0xFFFFL)))
```

キャストがいけないとき

さて，キャストというのは実際便利なもので，プログラムを楽に書けるようにしてくれるし，きちんとキャストするように心がければ，バグの混入を防げたり，プログラムの見通しも向上する．ところが，安易にキャストを使ってはいけない場合もあるということを知っているだろうか．

では，例として，次のようなバッファがあって，この中にいくつもの文字列をぎっしり格納するような場合を考えてみよう．

```
char buffer[BUFFER_SIZE];
```

通常のCの文字列('\0'で終了する)を詰めるのが充塡密度の点からいえば好ましいのだが，文字列を探すときに，それ以前の文字列をすべてスキャンして'\0'を見つけなければならなくなる．これは時間のかかる処理だ．

そこで，文字列の先頭に，その文字列が何文字(何バイト)なのかを記録しておく，という手段がある．文字列の大きさがいつもcharで表現できる大きさであるとは限らないので，大きさを記録しておくのにintを用いる

としよう.

これをDOS上のMS-Cを用いて実現すると，次のようなマクロを書くことになる.

```
#define next_string(p) \
    (*(int *)(p) == 0 ? NULL : (p) + *(int *)(p))
```

しかしこの方法では，移植性が失われてしまうのである.

先のマクロの引数であるpは，bufferの中のいかなる位置でも取り得る．つまり偶数アドレスや奇数アドレスも指せることは分るだろう．ところが，86系のマシンは奇数アドレスからワード(int)アクセスしても問題はないが，68系のマシンでそれをやると，プロセッサ例外(アドレスエラー)が発生してしまうのである．つまりこの場合は，intのポインタへのキャストを行って(アクセスして)はならないのだ(図4-1).

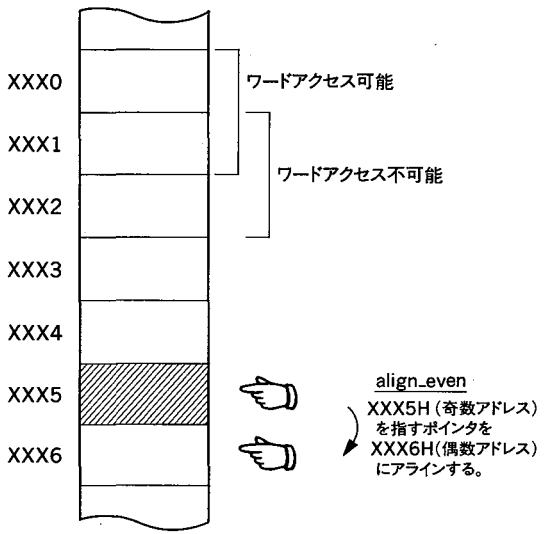
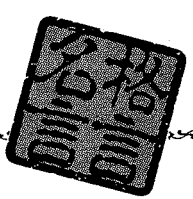


図4-1　アラインメントの考え方

これの解決策は2つある．2つのcharからintを合成するか，(長さを含めた)文字列が必ず偶数アドレスから始まるようにアラインメントするかである．後者を行うマクロを示しておこう．

```
#define align_even(p) \
        ((void *)( ((unsigned long)(p) & 1) \
                                ? (p) + 1 : (p) ))
```



ネストは育つ

＜解説＞

一応の完成を見たプログラムに仕様拡張などの改造を加え続けると，インデントが深くなりすぎてしまうこと．あるいは構造体のネスティングを覚えた「シロウト」が，際限もなく深くネストした構造体を使いたがる様子．

構造体を256倍使う

アルゴリズムとともにデータ構造の重要性がいわれるが，読者の方々はCの構造体を有効にきちんと使っているだろうか．構造体を有効に使いこなせばデータの抽象度が高まり，それなりにプログラムの読みやすさも向上するし，初心者からは，

なんだかよく分らないけど，すごいプログラムみたいだ

と思われて，尊敬されたりするはずだ(知らないけど)．Pascalでデータ構造が定義できるのは当然の話だが，アセンブラ(MASM)でさえもう何年も前からデータ構造を定義できるようになっているのだから，Cでプログラミングする我々としては，構造体くらい使えないと話にならない．構造体がよく分らない人は，K&Rを引っ張り出して勉強し直すようにしよう．また，たとえ構造体は多少使えても，共用体は使ったことがないような人も多いはずだ．そんな人のために，少しここでおさらいしておこう．

Cには共用体というデータ構造がある．これは単一のメモリ領域を複数の型で使い回すためのデータ構造だ．たとえば，次に示す共用体では，4バイトの領域をu.dwではlong型として，u.wではint型の配列として使えるのである．

```
union {
        long    dw;
        short w[2];
} u;
```

こういう単純な共用体は実際にはそれほど使う機会はないが，構造体の一部を共用体としたくなるケースはよくある．たとえば，次のような構造体を定義する場合があるだろう．

```
struct CMD {
        short func;
        short param1;
        union {
                long dw;
                short w[2];
        } param2;
} cmd;
```

この場合，cmd.func の値によって，共用体 param2 をどちらの型で使うかを決めたりする．つまり，

```
if (cmd.func == 0)
        cmd.param2.w[1] = 123;
```

といった感じで使うわけだ．ピリオドが 2 つ付いて，すこし長ったらしくなるが，これは仕方ない．

また，リスト 4-8 のように定義して，

```
if (cmd.func == 0)
        (struct CMD2)cmd.param2[1] = 123;
```

などとキャストしてアクセスするやり方もあるにはあるが，まあ，union を使ったほうが分かりやすいと思う．

```
struct CMD {
        short func;
        short param1;
        long  param2;
} cmd;

struct CMD2 {
        short func;
        short param1;
        short param2[2];
} cmd2;
```

リスト 4-8　union が嫌いな人の構造体定義

アラインメントに気を使うか？

構造体を使うときには，各要素のアラインメントに注意する必要がある．構造体のメンバというのはソースプログラム上では仲よく隣合っていて，いかにもメモリ中でも連続して格納されるように見える．ところが実際のメモリ中ではアラインメントのおかげでメンバとメンバの間にすきまが空いている場合も多く，これに気が付かないととんでもないバグが発生してしまうケースがある．逆に，メンバの間にすきまが空いていてもかまわないような場合には，アラインメントに配慮して，メンバを偶数番地に置くように気を使えば，メンバへのアクセス効率がよくなるのだ．まあ，8086 系の CPU では奇数番地からのワードアクセスも可能なので，あまり注意しなくてもよいともいえるが，やはり効率が違ってくるのだ．

とはいえ，実際にはアラインメントを無視せざるを得ない状況も多い．特にシステムコールで得られるデータをそのまま構造体として定義する場合などだ．たとえば，MS-DOSの非公開ファンクションコール32H(Get DPB Address)を使ってDPBを取得する場合，DPBはリスト4-9のような構造体として定義できる(ファンクション32HとDPBについての詳細はアスキー書籍編集部編著『MS-DOSプログラミングテクニック』を参照のこと)．

```
struct DPB {
        unsigned char   DriveNumber;
        unsigned char   UnitNumber;
        unsigned short  SectorLength;
        unsigned char   SectorPerAlloc;
        unsigned char   ShiftCount;
        unsigned short  ReservedSector;
        unsigned char   NumberOfFAT;
        unsigned short  DirectoryEntry;
        unsigned short  DataStartSector;
        unsigned short  DataSector;
        unsigned char   FATSector;
        unsigned short  DirectoryStart;
        void far        *ToDeviceHeader;
        unsigned char   MediaDescriptor;
        unsigned char   DiskChange;
        struct DPB far *ToNextDPB;
        unsigned short  CurrentCluster;
};
```

リスト4-9　DPBのデータを受け取る構造体

この構造体では，DirectoryEntry, DataStartSector, DataSector の 3 つが，奇数番地から始まるワードデータになる．ところが MS-C では，構造体の中のワード以上の大きさのデータが奇数番地から始まる場合，デフォルトではそれを偶数番地にアラインメントしてしまう．これは構造体アクセスの効率を上げるためなのだが，リスト 4-9 のような構造体では，それをやられてしまうとシステムコールで得られたデータと食い違いが生じてしまって，きちんと処理できない事態になる．したがって，こういう場合は−Zp オプションを指定して，構造体をパックする(偶数番地へのアラインメントをしない)ように指示する必要があるのだ．

ただし，MS-C ver.5.10 では，プラグマを使って個別のデータに構造体をパックするかしないかを指定できるようになっている．したがって，この構造体を定義しているファイルの先頭部分で，次のように指定しておけば，−Zp オプションをいちいち指定する必要がなくなる．

```
#pragma pack(1)          /* 構造体パック */
```

昔は構造体のパッキングを任せることができないコンパイラが多かった．つまり，きっちりパックしてしまい，奇数番地からワードアクセスのできないマシンに，それを強要する(アドレスエラーになる)ことがあったのだ．そのため，次のようにプログラマが，ダミーデータを入れて，アラインメントを手動で揃えることもあった．

```
struct DATA {
        char    a[5];
        char    dummy;
        long    b;
};
```

また，80386では32ビット(ロングワード)境界でもデータのアクセス速度が変わってくる．したがって，80386を意識すると，次のような構造が考えられる．

```
struct DATA {
        char    a[5];
        char    dummy[3];
        long    b;
};
```

無駄なデータが増えてしまうのを避けたければ，次のようにメンバの順序を入れ替えれば対処できる．

```
struct DATA {
        long    b;
        char    a[5];
};
```

しかしオプティマイズのすすんだ最近のコンパイラは，プログラマがアラインメントを調整しなくても，正しくかつ速いコードを生成するので，アラインメントに気を使う必要はない(かえってコンパイラを混乱させるかもしれない)．

構造体も長さが変わる

ところで，MS-C ver.6.00では次のような構造体が定義できる．

```
struct HEADER {
        char    name[8];
        short   attr;
        char    data[];
};
```

つまり，最後のメンバに限って，サイズを指定しない配列が記述できるのだ．これは，不定長データブロックのヘッダレコードを記述するのに使われる．すなわち，次のような記述で，不定長データブロックをメモリに読込んで処理する場合に便利なのだ．

```
struct HEADER *p;

fread(buf, size, count, fp);
p = (struct HEADER *)buf;
for (i = 0; i < 100; ++i)
        p->data[i] = 0;
```

ただしこの機能は，MS-C ver.5.10(およびそれ以前のバージョン)やANSI標準とは移植性がないので，注意してほしい．

ではここで，MS-C ver.5.10などでも使える，可変長構造体を実現するもう1つの技を考えてみよう．リスト4-10に示した構造体を見てほしい．

```
typedef struct _line_t line_t;
struct _line_t {
        line_t *next, *prev;
        u_char string[1];
};
```

リスト4-10　行格納用の構造体(その1)

この構造体の最後のメンバ，stringが今回の話題の中心である．大きさが1の文字列をメンバに持つstructなど，どんな意味があるのだろうか．その前に，このstructが一体どんな場面で使用されるものなのかを言っておかなければなるまい．それは，テキストエディタなのだ．

テキストエディタは，様々な長さの行を扱わなければならない．その行の長さに見合った大きさのメモリブロックをmalloc()で切り出してきて，その先頭のポインタを保持するような構造体を用意すればよい．そこで，リスト4-11のような構造体を考えてみた．

```
struct _line_t {
        line_t *next, *prev;
        u_char *string;
};
```

リスト4-11　行格納用の構造体(その2)

これはこれで目的にかなっているし，十分正しい．

しかし，これには問題がある．この構造体もmalloc()で切り出す(配列にしておくのは不経済だし，数が足りなくなるかもしれないから)のだとすると，1つの行をメモリにロードするのにmalloc()を2回呼び出さなければならないことになるのだ．また，malloc()は1つのメモリブロックに数バイトの管理情報を必要とするし，頻繁に呼ぶと主記憶の分断を招いてしまう．そこでリスト4-10に示しておいた構造体が登場するのである．

リスト4-10の構造体は次のように切り出せばよい．

```
#define alloc_line(len) \
        (line_t *)malloc((len) + sizeof(line_t))
```

こうすれば実質的に，望みの長さの行格納用バッファstringが得られるのだ．実際に使う場合には，たとえば次のようにすればよい．

```
line_t  *line;
line = alloc_line(strlen(buf));
```

そうして行を(ファイルやキーボードから)ロードした後は，たとえば次のような感じで通常の感覚で文字列にアクセスできるようになる．

```
c = line->string[i++];
```

こういうことができるのは，リスト4-10の構造体定義ではstringのサイズを1しかとらなかったものの，マクロalloc_lineのmalloc()のところで，構造体line_tの大きさと行のサイズを加えた領域を切り出したからなのだ．構造体定義のところで，

```
u_char  string[1];
```

となっているのがミソで，これがないとstringというメンバで領域にアクセスできないし，かといってMS-C ver.6.00のように

```
u_char  string[];
```

としてしまうと，MS-C ver.5.10などのコンパイラではエラーになってしまう．このstring[1]という技は，構造体定義でのstringのサイズに関わらず，その後のmalloc()の使いようによって可変長のstringを作り出すというアイディアなのである．

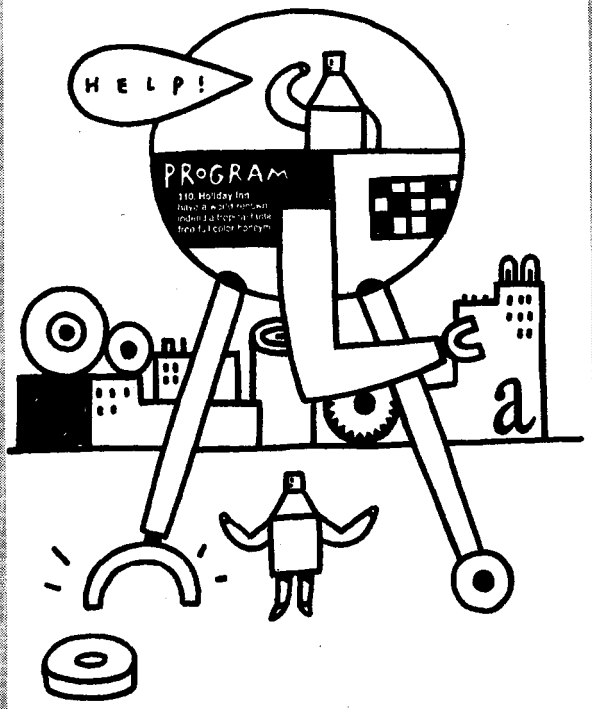
さて，大きさの変わるメンバは構造体の最後の(メモリに展開されたときに最も大きなアドレスを持つ)メンバでなければならず，しかもその構造体の中で唯一の可変長メンバでなければならない．なぜなら，構造体のメンバのアクセスは，構造体の先頭からのオフセット(構造体の定義時に決まる)によるからだ．どうしても2つ以上のメンバを可変長にしたいのなら，通常の構造体のアクセス方法を諦めなければならない．つまり構造体にする意味はないのである．

案ずるより生むが易し

＜解説＞

初心者の場合，インタプリタ形式の言語処理系を使うときに心の中で唱える勇気の呪文．上級者の場合は，プログラムのドキュメントを後回しにするときの言い訳．

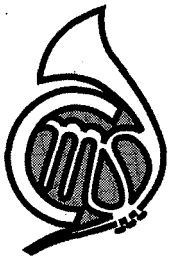
ライブラリ活用のヒント



年齢を重ねて(あるいは実力があって)人を使う立場になると，部下の性格をよく把握することが大切になる．会社には，会議で流暢に長々としゃべる割には小回りや機転のきかないCOBOL型人間や，頭は切れるし話も簡潔にできるのだが，ちょっといい加減なところのあるC型人間，バカでいつも何をいっているのか分らないが短時間の力仕事だけはめっぽう得意なBASIC型人間，上品で優雅で美しいが，汚い仕事にはそっぽを向いてしまうPascal型オフィスお嬢様など，さまざまな性格の部下がいるのだ．こうした人々を適材適所に使いこなすのが，おそらく優れた管理職になる条件だろう(知らないけど)．

それと同じように，Cで優れたプログラムを書きたいと思ったら，ライブラリ関数について十分に理解しておかなければならない．といっても単に機能を知っていれば済むわけではなく，個々のライブラリ関数の性質やメリット/デメリットをよく把握し，状況に応じて細かく使い分ける必要がある．

ここでは主に入出力関係とメモリ操作関係の関数を取り上げ，それぞれの特質を調べてみよう．



入出力関数

まず，表5-1の表を見ていただきたい．この表は，色々な場合にread()とwrite()が入力や出力をどう扱うかを示したものだ．表の中に，わざと空欄にしてある部分があるので，そこに何が入るのか，自分で考えてみてほしい(解答はこの節の最後に掲載する)．

● read()の場合

入力元	モード指定	バイト数指定	入力文字 ^M	^J	^Z
キーボード(cooked)	O_TEXT	n	\n (EOL)	(CONT)	(EOF)
		1	\n (EOL)	(CONT)	(EOF)
	O_BINARY	n	\r\n (EOL)	(CONT)	行頭(EOF) それ以外\x1a
		1	\r\n (EOL)	(CONT)	行頭(EOF) それ以外\x1a
キーボード(raw)	O_TEXT	n*	①	\n	(EOF)
		1	\n	\n	(EOF)
	O_BINARY	n*	\r	\n	\x1a
		1	\r	\n	\x1a

入力元	モード指定	バイト数指定	入力文字 \r	\n	\r\n	\x1a
ファイル	O_TEXT	n	\r	\n	\n	(EOF)
		1	\r	\n	\n	(EOF)
	O_BINARY	n	\r	\n	\r\n	\x1a
		1	\r	\n	\r\n	\x1a

● write()の場合

出力先	モード指定	バイト数指定	出力文字		
			\r	\n	\x1a
ディスプレイ(cooked)	O_TEXT	n	CR	CR-LF	x
	O_BINARY	n	CR	LF	x
ディスプレイ(raw)	O_TEXT	n	CR	②	CLS
	O_BINARY	n	CR	LF	CLS

出力先	モード指定	バイト数指定	出力文字		
			\r	\n	\x1a
ファイル	O_TEXT	n	\r	③	\x1a
	O_BINARY	n	\r	\n	\x1a

■ この表の見方 ■

・(EOL), (CONT), (EOF)は文字コードではなく，それぞれ「行の終わり」,「行の継続」,「ファイルの終わり」として処理されることを表わす．
・\r, \n, \x1a は文字コードで，それぞれ「0x0d」,「0x0a」,「0x1a」のことである．
・CR, LF, CLS は文字コードではなく，それぞれ「復帰」,「改行(ラインフィード)」,「画面消去」の画面制御を表わす．
・*印は n(>1)バイトの読出し要求をしても，1つキーが押されただけでリターンする．
・raw モードのキーボードを O_BINARY モードで1バイト読み出したときの挙動は getch()と同じである．
・raw モードのディスプレイに O_BINARY モードで書き込んだときの挙動は putch()と同じである．

表 5-1　read()と write()の振舞い

さて，MS-C には非常に多くの種類の入出力関数がある．一見すると，似たような用途のために，どうして何種類も関数があるのか疑問に思うはずだ．そこで，主な入出力関数を階層別に表にしてみた(表 5-2)．

ストリーム入出力 (高水準入出力)	fgetc getc (m)	gets	printf	fputc putc (m)	fread	scanf	fwrite	puts
低水準入出力					read		write	
コンソール入出力	getch getche	cgets	cprintf	putch		cscanf		cputs
DOSシステムコール					_dos_ read		_dos_ write	

(m)はマクロとして実現されている

表 5-2　入出力関数の階層別一覧

この表に示したとおり，入出力関数は，ストリーム入出力，低水準入出力，コンソール入出力，DOS システムコールの 4 つの階層に分類される．この階層分けには歴史的にきちんとした根拠があって，それぞれの目的も使い方もはっきりと区別できる．

ストリーム入出力関係の関数は C 言語本来の伝統的な高水準入出力を実行する．一応 OS には依存しないので，一番移植性が高い．そしてこの高水準入出力を実現する元になっているのが低水準入出力だ．open()，read()，write()などの低水準入出力関数は，どの処理系でも用意されているので一見標準的な C の関数に思えるかも知れないが，実際には OS 依存で，細かい動作は OS のカーネルの機能に深く関わっている．こうした関数は本来は UNIX のものであり，単に UNIX のシステムコールを呼び出すインターフェイスとして機能するだけなのだ．一方 MS-DOS(ver.2.*x* 以降)にも UNIX の 1 バージョンである XENIX 互換のシステムコールがあるため，MS-DOS 用の C の処理系でもこうした低水準関数をそのままサポートできたのである．

コンソール入出力関数の場合には，MS-DOS が引きずっている CP/M 互換のファンクションコールを呼び出すようになっている．だから，関数の仕様はストリーム入出力関数の仕様と似てはいるが，実現方法はまったく違っているのだ．また，_dos_read()など，MS-DOS のシステムコール

を直接呼び出す関数群は，当然ながら低水準入出力関数と非常によく似ている．

さて，このように歴史的な経緯によって関数がいろいろあるのだが，それぞれの関数にはもちろん向き不向きがある．といっても，基本はやはりストリーム入出力だ．ストリーム入出力でうまくいかないような処理は，たいていは本来アセンブラで書くべき部分だろう．それなのに，あれこれ駆使すればCで書けてしまうのが，Cのいい加減なところというか，偉いところなのだ．

ストリーム入出力のメリット/デメリット

ストリームは何らかのデバイスに結び付いたデータの入出力先で，たとえば標準入力(stdin)や標準出力(stdout)はストリームの例である．ファイルをオープンすればファイルもストリームになり得るが，ストリーム入出力の大きな特徴は，ファイルのアクセスにFCBやファイルハンドルを使うのではなく，

ファイル構造体を使う

という点にあるのだ．ファイル構造体はstdio.hで定義されており，バッファへのポインタ，アクセスモードを示すフラグやファイルハンドルなどが入っている．低水準入出力などの場合とは違って，ファイルとのデータのやり取りは，このファイル構造体へのポインタ(ファイルポインタ)を利用して行われる．

ストリーム入出力のもう1つの大きな特徴として，

ライブラリレベルでのバッファリングを行う

という点があげられるだろう．このバッファリングはカーネルレベルでのバッファリングとは別物で，Cのライブラリが自前で提供しているものだ．バッファリングには，ファイル構造体で定義されているバッファが利用される．

一般に，ファイルアクセスを高速化するためには何らかのバッファリングメカニズムが不可欠だ．普通の処理をする場合には，ストリーム入出力を使えばバッファリングしてくれるので，それで十分だろう．しかし，標準入力から1文字受け取ってすぐに処理したい場合(後述)など，バッファリングを禁止する必要のあるときもある．その場合には，次のようにすれば，バッファリングが行われないようになる．

```
setvbuf(stream, NULL, _IONBF, 0);
```

ただ，ファイルではなくデバイスと入出力を行いたいとか，高速化のためにプログラムで自前のバッファリングをするなどの場合には，ストリーム入出力を使わずに低水準入出力を使った方が効果的な場合が多い．

低水準入出力はシステムコールのレベルに近く，ファイルアクセスにファイル構造体ではなくファイルハンドルを利用する．デバイスからの直接入力や，自前で大規模なバッファリングを行いたいときなど，低水準入出力を使えば非常に自由度の高い処理が行えるし，高速化できる可能性もある．しかし，

自由度の高さ

と

自分であれもこれもやらなきゃいけなくて大変

というのは表裏一体で，豊富な関数があるストリーム入出力と違って，何でもかんでもread()やwrite()だけでやろうとすると非常に骨が折れる．その点ストリーム入出力ならば楽だし，保守もしやすいし，移植性も高まる．が，どちらがいいかということは，プログラムの目的や文脈にもよるので一概にいえないし，プログラマが柔軟に判断するほかない．

親切なread()

MS-Cのread()にはテキストモードとバイナリモードがある．バイナリモードでは，MS-DOSのファンクションを普通に使ってファイルハンドルから読み込むだけだが，テキストモードでは，UNIXとの互換性をとるためにけっこう複雑な処理をしている．具体的には，行末コードのCR，LFのペアを1つのLFに変換することと，EOF(CTRL-Z)の処理である．しかし，ただCR，LFをLFに変換するという単純な話では済まないから問題なのだ．

バッファの最後にCRがきたときに，read()は次の1バイトを先読みする．そしてそれがLFかどうかを判断し，LFならばCR，LFを1個のLFに変換している．rawモードになっているデバイスから読んでいる場合には，先読みするわけにはいかないという問題もある．

こうした問題に対処するため，read()は処理のたびに何回もMS-DOSのファンクションを呼び出していて，そのオーバーヘッドがばかにならないのである．コードサイズも大きくなるし，わざわざ低水準入出力を使うメリットが減少してしまうのだ．こんなときだけは，

何もしないread()がほしい

と思ったり，ほかの処理系の貧弱なライブラリがうらやましくなってしまう(?)．テキストモードだったらCRが単独で入ってくることもないだろうし，rawモードのデバイスをテキストモードでオープンするなんてこともほとんど考えられないはずだ．だったら，CRがきたらとにかく次の1バイトを読み捨ててLFを返すようなテキストモードもあればよかったのに，と思ってしまう．

このread()のオーバーヘッドへの対策として，ちまちま読まないで一気に大量に読み込むという手がある．そうすれば，読込みの回数が減る分だけ，バッファの最後にCRがくる確率も減るはずだ．あるいは，そもそもテキストモードなんていうのは使わないで，根性を出して全部プログラマが自前で処理するのがいいだろう．

また，テキストモードの弊害というか，困る点として，CTRL-Zがくると読み込みを終了してしまうということがある．入力元がコンソールの場合には，CTRL-Zがきたときに EOF だと見なさなければならないので仕方ないが，ファイルの場合にはCTRL-Zのあとにも意味のある情報が続いているときがあるのだ(DOS上のCの処理系の中には，CTRL-Zの後ろにfseekできてしまうものもある)．これを救済してやろうと思うと，非常に面倒な処理が必要になる．この辺，テキストモードは一見親切ではあるけれど，思わぬ落し穴もあるということを知っておいた方がよい．

コンソール入出力とMS-DOSシステムコール

MS-Cの入出力関係の関数群は，機能は確かに豊富である．だが，そんな機能はどうでもいいから，とにかくスピードを優先させたいときはどういう関数を使ったらよいだろうか．

まず，コンソール入出力関係では，次のような関数がある．

cgets()，cputs()，cprintf()，cscanf()

これらは，機能的には関数名の先頭に〝c〟の付かないバージョンと一緒だが，昔は，リダイレクトされていようといまいと，入出力先が必ずコンソールになった．MS-DOSのもっとも低レベルなファンクション(いわゆるCP/Mコンパチブルコールってやつ)を使っていたのだ．そのため，これらのコールを使用すると，今でもコンソールによって入出力を行えると思われがちだが，ver 2.*x*以降ではシステムコールがリダイレクトの影響をうけるために，コンソールから入力できるとは限らない．結局，〝c〟の付かないバージョンと違う点は，CTRL-JやCTRL-Zを特殊解釈しないという点に絞られる．

これの仲間には次の関数も含まれる．

getch()，getche()，putch()，ungetch()

デバイスに対し〝raw〟な入出力を行うとき，デバイスドライバをrawモードに設定して，(バイナリモードのハンドルを使って)readするという手段もあるが，これはデバイスドライバがrawモードをサポートしていなければ話にならない．またこの方法でwriteすると，PRINT.EXEがうまく動かないこともあり問題だ．

そこでgetch()などの登場であるが，これではEOFを検出できない(CTRL-Zを返される)．そのため，自前でEOF文字の検出を行うのだが，ブロックデバイスの場合は長さによってファイルの終わりを検出するものなので，EOF文字があるとは期待できない．つまりハングアップする可能性があるのだ．

回避するには，isatty()でデバイスのタイプを調べ，キャラクタデバイスのときだけread()を使用するようにする．キャラクタデバイスの場合は，通常は人間がキーボードを打つのを待つわけだから，いつかはEOF文字が入力されると期待してもよい．

さらに，_dos_～の関数群もある．入出力関係に限っていえば，MS-C ver.5.10では次のような関数が用意されている．

_dos_close()，_dos_creat()，_dos_creatnew()，_dos_open()，
_dos_read()，_dos_write()

このような関数は，別にもっともらしいことなど何もせずに，引数をMS-DOSのファンクションに渡して単純にコールするだけだ．ただ，低水準入出力とあまり変わらないので，低水準入出力関数をread()のところで述べたような問題点に注意して使えばよいので，_dos_～関係はあまり使う必要性がないように思う．

大体，本当にスピードが優先されるような場合なら，こういう関数など使わないで，自分でアセンブラで書いた方がいいのではないだろうか．それに，移植性のない関数がソースプログラム中に散らばっているよりも，アセンブラを使って移植性のない部分だけモジュール化した方が，確実にプログラムの見通しがよくなるはずだ．以前ならアセンブラで書いていたルーチンをCで簡単に呼び出せるようになったという以外に，それほどメリットはないだろう．

キーから1バイトだけ読み込む

標準入力を利用して，コンソールから1バイトだけ読んで何かしたいという場合があるだろう．ゲーム中でキーによってキャラクタの動きを制御したいとか，スクリーンエディタでユーザーからコマンドを受け取りたいとか，メニューを出して項目を選ばせたいというようなとき，タイプしたキーが画面にいちいちエコーバックされたり，ユーザーがリターンキーを打たないとプログラム側でキーを受け取れなかったりするのでは，まどろっこしくて仕方がない．

こうなってしまうのは，コンソールデバイスを通常のモード(cooked モード)のままで使っているからだ．コンソールデバイスでの cooked モードの特徴をあげると，次のようになる．

・読み出すバイト数に関わらず，行単位で入力が行われる
・入力した行がバッファリングされる
・タイプした文字がエコーバックされる

これはこれで普通は便利なのだが，前述のようなゲームなどの用途には，はっきりいってドン臭くて使いものにならない．そこで，こういうときにはデバイスの raw モードを利用するのが普通だ．raw モードはその名のとおり「調理されて(cooked)いない生の(raw)」モードで，次のような特徴がある．

・指定のバイト数を受け取ると，プログラム側に戻る
・制御用の文字(CTRL-Z など)が変換されない
・タイプした文字がエコーバックされない

上記のようなゲームとかの場合には，この raw モードを利用するのが普通である．デバイスを raw モードにするには，UNIX ならば ioctl() という関数を使ってやればよいが，MS-C のライブラリには用意されていないので，intdos() などを使って，MS-DOS のファンクションコールを呼び出して実現することになるはずだ．具体的には，**リスト 5-1** のような関数を作ればよい．

```
#include        <stdio.h>
#include        <dos.h>
#include        <io.h>

int
fsetraw(stream)
FILE    *stream;
{
        union   REGS    inregs;
        union   REGS    outregs;

        if (!isatty(fileno(stream)))
                return  -1;
        inregs.x.ax = 0x4400;
        inregs.x.bx = fileno(stream);
        intdos(&inregs, &outregs);
        inregs.x.dx =
                outregs.x.dx & 0x00ff | 0x0020;
        inregs.x.ax = 0x4401;
        intdos(&inregs, &outregs);
        return outregs.x.cflag;
}
```

リスト5-1　デバイスをrawモードに設定する関数

ただし，この関数ではBREAKを殺していないので，CTRL-Cが入力されたときにプログラム側でそれを利用したい場合には，ファンクション33Hを使ってBREAKを無効にしておく必要があるだろう．

さて，1文字受け取ってすぐに何かしたい場合には，この関数だけでは不十分で，プログラムの最初の方で前述のsetvbuf()を使ってバッファリングを禁止しておかなくてはならない．こうしてめでたくコンソールデバイスをrawモードにできるようになったはずだが，何の因果かrawモード

をサポートしていないタコなデバイス(一部の仮名漢字変換FEPなど)もあるので注意してほしい．rawモードは使わずに，ファンクション06H/07Hあたりを使うのが無難なのかも知れないが，それよりそんなデバイスは

捨ててしまうのが一番

である．

もっとも，rawモードだとかcookedモードだとかバッファリングを禁止してどうのとかいうのは，確かに面倒に違いない．どうせMS-DOSの場合には，1文字入力をしたいのはコンソールからだろう．だったら，いちいちストリームやファイルハンドルなんか気にしないで，もっとも低レベルなgetch()やgetche()を呼び出すのが一番楽だと思う(それぞれ，ファンクション08H/01Hを利用している)．

やたらとでかいprintf()

さて，数あるライブラリ関数の中でも，printf()は何かを表示したりするときに便利なので，使われる頻度が非常に高い．K&RとかCの入門書でも，最初に出てくるのはprintf()を使ったプログラムだったりする(リスト5-2)．

```
main()
{
        printf("hello, world.\n");
}
```

リスト5-2　printf()版 "hello, world"

そんなわけで，何でもかんでも手軽にprintf()を使っている人も多いのではないだろうか．

ところが，printf()というのはとんでもなく無駄な関数なのだ．リスト5-2をMS-C ver.5.1でコンパイルして，サイズを調べてみた．

```
-a--w     7283 Jul 11 1990 20:49 printf.exe
-a--w      297 Jul 11 1990 20:49 printf.obj
```

一方，同等のプログラムを今度はputs()を使って作成してみる(リスト5-3)．

```
main()
{
        puts("hello, world.");
}
```

リスト5-3　puts()版 "hello, world"

これも，コンパイルしてサイズを調べてみよう．

```
-a--w     5245 Jul 11 1990 20:49 puts.exe
-a--w      295 Jul 11 1990 20:49 puts.obj
```

これだけのプログラムなのに，printf()を使ったプログラムとputs()版のプログラムでは2Kバイト以上のサイズの差が出てしまった．ついでに，putchar()とwrite()を使ったバージョンも調べてみよう(リスト5-4，リスト5-5)．

```
#include        <stdio.h>
char    text[] = "hello, world.\n";
main()
{
        char    *outp = text;

        while(*outp)
                putchar(*outp++);
}
```

リスト 5-4　putchar()版 "hello, world"

```
char    text[] = "hello, world.\n";
main()
{
        write(1, text, sizeof(text) - 1);
}
```

リスト 5-5　write()版 "hello, world"

やはりサイズを調べてみると，

```
-a--w    4527 Jul 11 1990 21:05 putchar.exe
-a--w     396 Jul 11 1990 21:05 putchar.obj
-a--w    2763 Jul 11 1990 21:05 write.exe
-a--w     319 Jul 11 1990 21:05 write.obj
```

なるほど，どんどん小さくなる．write()版と比べると，printf()版は 2.5 倍もサイズが大きくなっているのがわかるはずだ．

とはいえ，小さくなるからといって，ここまでやるのはやり過ぎかも知れない．こんな調子でやるつもりなら，最初からC言語なんかでプログラムを書かない方がいいに決まっているからだ．しかし，とにかく，printf()がとてつもなくでかい関数だというのは判明した(プログラムのサイズをいかに小さくするかという問題は次の章でも扱うので，そちらもあわせて参照してほしい)．

考えてみれば，Cのライブラリというのは，どういう状況でも使えるようになるべく一般的に書かれているからこそ存在意義があるわけで，プログラマが自分でどんどん低レベルな部分まで面倒を見るようにすればプログラムが小さくなるのは当然だし，逆に，汎用的なライブラリを使えばある程度サイズが大きくなるのは仕方ないことなのだ．しかし，それにしてもprintf()はでかい．ライブラリのマニュアルを見てほしい．printf()では，かなり複雑な出力フォーマットが指定できるし，それに応じて山のようにフラグだのなんだのが用意されている．これを処理するのはいかにも大変そうだし，オブジェクトのサイズが大きくなるのも当然だと思わず納得してしまう．そもそもprintf()はライブラリ関数なので，引数として与えられたフォーマット文字列を実行時に解釈するしかない．Pascalのように，出力フォーマットの指定が言語処理系の機能になっていれば，出力ルーチンを小さく速くできたはずだ．

printf()は確かに便利だが，簡単な文字列の出力くらいだったら，まあwrite()はやり過ぎとしても，puts()あたりを使うのが一番だろう．しかし，当り前の話だが，プログラムの別の部分で複雑な出力を行っていて，そこでprintf()を使っているなら，文字列出力の部分だけをputs()に置き換えてもプログラムのサイズが減少するわけはないので，プログラム全体を見通してprintf()を使うかどうかを決める必要がある．

また，考えようによっては，printf()をputs()に置き換えたところで2Kバイト位しか小さくならないなら，サイズが50Kバイトのプログラムではたいした影響がないともいえる．ただ，

使いもしない無駄なコードが勝手にリンクされるのは嫌だ

というのは，誰しも思う(?)ことだろう．確かに

それならCじゃなくてアセンブラで全部書けばいいのに

というのはもっともな話で，「はい，どうもすいません」と謝るほかはないが，Cだろうと何だろうと，プログラムの見通しが悪くならない程度に工夫してプログラムのサイズが小さくなるなら，それに越したことはないと思う．

printf()の変な引数

さて，printf()(およびそのファミリー)にはサイズがでかいというほかに，

可変長引数をとる

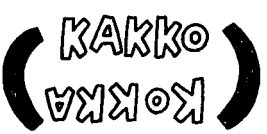
という特徴がある．引数の数が一定していない関数は気持ち悪いという見方もあるが，これが実現できたのは，Cの引数のスタックへの積み方のおかげである．というより，可変長引数を実現するために，効率を多少犠牲にしたやり方を選択したというのが正しい．

引数をスタックで受け渡すマシンのCでは，関数を呼び出す際に，最後の(ソースプログラムで一番右の)引数からスタック上にプッシュし，最初の引数が一番上になるようにスタックに積んでいく．これは次の関数，

```
func(a, b)
int a, b;
{
        return (&a + 1 == &b);
}
```

が〝真〟を返すことを保証する．つまり引数の左から順にアドレスが(その引数の大きさだけ)大きくなっていくのである(aがcharのとき，&a + sizeof(char) == &bとは限らない，結局は処理系に依存する*1)．

そして，呼び出した関数が戻ったときに，呼び出し側で引数を取り除くようになっている．したがって，呼び出された関数の方では，実際に引数がいくつあるのか実行時まで分らなくても正常に動作できるのだ．これが呼び出された関数の方でスタックポインタを調整して引数を取り除かなくてはならない場合(pascal呼び出しなど)には，速度的には若干有利なのだが，関数をコンパイルする時点で引数の型や数が分っている必要があるので，printf()のようなライブラリ関数の実現が困難になってしまう(リスト5-6参照)．



*1もしも，下方伸張のバッファ(スタック)ではなく，上方伸張のバッファを使って引数を受渡しするようなマシンだと，最初(ソースプログラムの一番左側)の引数から積んでいかなければならない．こんなわけで，引数の評価順序は処理系に依存してしまうのである．

```
int     cdecl   a(int, int); ←普通のCの関数
int     pascal  b(int, int); ←pascal呼出し

関数a(4, 2)の呼出し
mov     ax,2
push    ax
mov     ax,4
push    ax
call    _a
add     sp,4 ←呼出し側で引数を取り除く
...

関数b(4, 2)の呼出し
mov     ax,4 ←最初の引数からスタックに積む
push    ax
mov     ax,2
push    ax
call    B
...
関数b()の定義
        PUBLIC  B
B       PROC NEAR
push    bp
mov     bp,sp
...
pop     bp
ret     4 ←関数側で引数を取り除く
```

リスト 5-6　C 呼び出しと pascal 呼び出し

ただし，Cでの引数の渡し方のおかげで可変長引数の関数が実現できたといっても，printf()のようにあらかじめどんな型の引数が渡されるか分らないような関数では，移植性が問題になる場合がある．それでも移植性を何とか保証しようという苦肉の策が，stdargパッケージとかvarargsパッケージとか呼ばれるマクロだ．MS-Cにもstdarg.hとvarargs.hという2つのヘッダファイルが付属していて，どちらかをインクルードして，vprintf()とかvfprintf()とかのファミリーと一緒に使ってやればよい．

ところが，お茶目なことに，stdarg.hとvarargs.hで定義されているマクロは互いに細かい点が違っていて，これ自体移植性がないのだ．varargsの方はSystem VやXENIX系で，これまでも広く使われてきたが，ANSIの標準ではstdargで定義されているマクロの方を使わなくてはならないことになっている．まあどっちにしてもマクロなので，気に入らなければstdarg.hやvarargs.hを研究して自分で書いて，ヘッダファイルとして用意しておけばよいだろう．

scanf()なんか使わない

先日，岩手県から家出して上京してきた少年たちが補導されたという事件が新聞に報道されていた．これだけなら別に普通の記事だが，その新聞報道によると，彼らは

岩手のシンナーよりも東京のシンナーの方がうまい

という噂に誘われて家出してきたという点が何とも画期的(?)だった．このように，人間の思い込みというのは，時としてとんでもない結果をもたらすものなのだ．

Cのライブラリには，scanf()という関数がある．一見すると，入力を行うprintf()みたいに思えるが，単純にscanf()を信用して便利そうだからといってホイホイ使うと，さすがに補導はされないものの，えらい目にあってしまう．

特に，人間からの入力を直接scanf()で処理するのは厳禁だ．人間は必ず間違いを起こすのにscanf()にはその間違いを訂正する能力がなく，致命的な結果を招くケースが多いのである．たとえばリスト5-7を見てほしい．このサンプルプログラムは，標準入力から日付，項目，金額を読み込む．

```
#include <stdio.h>

main()
{
    int month, day, amount;
    char desc[128];

    while (1)
    {
        if ( scanf("%2d/%2d %s %5d", &month,
                    &day, desc, &amount) == EOF )
        {
            break;
        }
        printf("Month: %d, Day: %d, ",
                                    month, day);
        printf("Descr: %s, Amount: %d\n",
                                    desc, amount);
    }
}
```

リスト5-7　scanf()を使ってデータを読み込むプログラム

では，これをコンパイルしてできあがったプログラム(INPUTD.EXE とする)を実行してみよう．

```
A>inputd
5/10 test 12443
Month: 5, Day: 10, Descr: test, Amount: 12443
```

上では，入力データがscanf()の書式文字列に指定したとおりなので，正常に実行できた．ところが，次の例のように入力を少しでも間違えると，表示が止まらなくなってしまう．

```
A>inputd
5.10 test 12312
Month: 5, Day: 10, Descr: test, Amount: 12443
Month: 5, Day: 10, Descr: test, Amount: 12443
Month: 5, Day: 10, Descr: test, Amount: 12443
.....(止まらない)
^C
A>inputd
5/10 test 123,434
Month: 5, Day: 10, Descr: test, Amount: 123
Month: 5, Day: 10, Descr: test, Amount: 123
Month: 5, Day: 10, Descr: test, Amount: 123
.....(止まらない)
```

また，次の実行例を見てほしい．

```
A>inputd
001/10 test 32132
Month: 0, Day: 249539148, Descr: , Amount: 249539140
Month: 1, Day: 10, Descr: test, Amount: 32132
1/020 test 323
Month: 1, Day: 2, Descr: 0, Amount: 32132
Month: 1, Day: 2, Descr: 0, Amount: 32132
Month: 1, Day: 2, Descr: 0, Amount: 32132

.....(止まらない)
```

こんな風になってしまうのはscanf()のせいには違いないが，scanf()の仕様上，データの入力とscanf()自身によるデータの取り込みを同期させられないので，バッファに残ってしまっているデータの取り扱いが非常に難しいのだ(たとえば"\n"が入力された以降のデータを無条件にクリアしたりしないのが，scanf()の仕様)．人間が常に正しいデータを入力するなどと信じるプログラマは多分いないと思うので，scanf()では人間からの入力を直接受け取らずに，

前もって正しいと分っているデータの並びを切り分ける

ためだけにscanf()を使うべきだ．

ところで，scanf()に似た関数にsscanf()というのがあって，こちらはまだscanf()よりは安全に使える．先ほどのプログラムを，sscanf()を使ってリスト5-8のように書き換えてみた．

```
#include <stdio.h>

main()
{
    int month, day, amount;
    char desc[128];
    char tmpp[256];
    while (1)
    {
        if (gets(tmpp) == NULL) break;
        if (sscanf(tmpp, "%2d/%2d %s %5d",
                &month, &day, desc, &amount) != 4)
        {
            continue;
        }
        printf("Month: %d, Day: %d, ",
                                    month, day);
        printf("Descr: %s, Amount: %d\n",
                                    desc, amount);
    }
}
```

リスト5-8　sscanf()版データ入力プログラム

これを実行してみる.

```
A>inputd2
5/10 test 12343
Month: 5, Day: 10, Descr: test, Amount: 12343
5.10 test 12343   ┐
01/3 test 32323   ├ 間違った入力
1/0200 test 32323 ┘
^Z
```

こちらのプログラムの場合には，scanf()版とは違ってプログラマが自分でバッファの操作を行っているので，一応安全に実行できる．入力が間違っているときに適切なメッセージを出力したりすれば，人間が入力するときでも安全性が高まるだろう．

まあ，なんだかんだいっても，我々の回りでは誰もが

scanf()？　そんなもん普通使わないでしょ

と言い切っている．いずれにせよ，便利そうだからといってこんなわけの分らないライブラリ関数に頼らないで，入力データぐらいプログラマが自力でまともに切り分けるのがやはり一番正しいやり方だろう．

表 5-1 の空欄部分の解答：

(1) \r[*2]
(2) CR-LF
(3) \r\n

[*2] (1) はなぜか'¥n'にならない(MS-C の場合)．これははたして仕様だろうか，バグだろうか？

メモリモデルとメモリ管理

MS-C は 8086 をターゲット CPU とする C なので，8086 のアーキテクチャに強く依存した部分も持っている．それがメモリモデルだ．

良く使われるメモリモデルは，スモールモデルとラージモデルである．これらのメモリモデルはそれぞれポインタとして near ポインタと far ポインタを使用している．ここで少しおさらいをしておこう．

よく知られているように 8086 はセグメント単位でメモリを管理している．セグメントは 16 ビットの値で識別し，セグメントの内部は 16 ビットのオフセットでアドレスする．もしもプログラムが 1 つのセグメントしか持たないのであれば，全てのアドレッシングに 16 ビット値を用いれば良い．これが near ポインタである．これに対し，プログラムがいくつものセグメントを持っている場合には，どこか特定のアドレスにアクセスするには，セグメントを識別する 16 ビット値とその内部オフセットの 16 ビット値(near ポインタに相当)の組である 32 ビット値を用いなければならない．この 32 ビット値が far ポインタである．

明らかに far ポインタを格納するために near ポインタの 2 倍のメモリを必要とするし，アクセス速度も遅くなる．しかし near ポインタに比べて圧倒的に大きなメモリ空間をアドレスできるというメリットがある．

コードセグメントもデータセグメントも 1 つずつ，つまりどちらにも near ポインタが使用できるようなプログラミングモデルをスモールモデルという．そしてコードセグメントもデータセグメントも複数あって(1 つでも構わない)，どちらも far ポインタを用いるようなプログラミングモデルをラージモデルという．

near/far ポインタに気をつけろ

さておさらいはこれくらいにして，問題点の指摘に入ろう．MS-C では near ポインタと far ポインタを混在して使用できる．しかしライブラリ関数がこれについてこない場合があるのだ．

MS-C には slibc *.lib とか llibc *.lib とか先頭の 1 文字にメモリモデルを示すライブラリがあって，デフォルトのメモリモデル(−A で指定する)に従ってどのライブラリをリンクするかを決定している．ここで，スモールモデルのライブラリ関数に far ポインタを渡す(あるいはその逆)とどうなるかを考えてみよう．

リスト 5-9 のサンプルルーチンを見てほしい．これは，PC-9801 のテキスト画面を退避させるルーチンで，スモールモデル用に書いてある．

```
/*
 *  動かない例
 */
#define VRAMSEG (0xa000)
#define VRAMSIZE (0x4000)

#include <dos.h>
#include <memory.h>
#include <malloc.h>

char *savedata()
{
    far char *vram;
    char *savep;

    FP_SEG(vram) = VRAMSEG;
    FP_OFF(vram) = 0;
```

```
    if ((savep = malloc(VRAMSIZE)) == NULL)
        return NULL;
    return (char *)memcpy(savep, vram, VRAMSIZE);
}
```

リスト5-9　正常に動作しない画面退避ルーチン

この場合のmemcpy()はスモールモデルのmemcpy()であり，こいつはポインタといえばみなnearポインタだと思い込んでいる．ここで，savepをmalloc()した結果が0x1234だったとする．この値は(スモールモデルでは)1つしかないデータセグメント内でのオフセットだ．すると，memcpy()を呼出す部分では，スタックには

```
0x1234 0x0000 0xa000 0x4000
```

が積まれる．しかしmemcpy()は，パラメータとして2つのnearポインタと1つの16ビット値を渡されたと思い込んでいるから，なんとデフォルトデータセグメントのオフセット0x0000からの0xa000バイトのデータを，オフセット0x1234にコピーしてくれちゃうのである．

これでは目茶苦茶だ．それでもよっぽど運がよければ単に意味のないコピーが起こるだけで済むが，たいていの場合は，カタストロフィーが訪れることになるだろう．

最近のCでは関数に渡す引数の型を宣言しておけば，自動的に型変換が行われるようになっている．その場合は，カタストロフィーというほど致命的な結果とはならないだろうが，memcpy()に渡されるパラメータは

```
0x1234 0x0000 0x4000
```

となり，やはり正しく動かないことには変わりがないのだ．

MS-C では near ポインタや far ポインタを手軽に使えて便利なのだが，ライブラリ関数に渡す場合には，このように思わぬ結果を生じることもあるので，くれぐれも細心の注意が必要である．

セグメントと far ポインタの微妙な関係

8086 はセグメント単位でメモリを管理しているが，このセグメント単位というのが非常にやっかいな問題を抱えている．その代表選手が，far ポインタのふるまいによる問題である．

malloc() という関数は，指定した大きさのメモリ領域を割り当ててくれる．しかし，8086 のセグメントアーキテクチャを尊重すればこのメモリ領域は最大でも 64K バイト未満だというのに，それよりさらに小さいメモリ領域が 1 つのセグメントに収まり切らないなんていうことがあったら大変である．「1 つのセグメントに収まり切らない」とは，オフセット部の 16 ビットだけで線形にアクセスできなくなるという意味だ．一体どういうことか，これだけでは何がなんだか分らないかも知れない．では例として，ラージモデルの malloc() (_fmalloc()) でメモリを割り当て，far ポインタを返してもらったときのことを考えてみよう．

_fmalloc() で 1024(0x400) バイトを確保して，0x00f0：x0f000 という far ポインタが返ってくるなら，そこからの 0x400 バイトはオフセット部の加算だけで線形にアクセスできるので，とってもグッドである．だが，もしここで 0x0000：0xff00 というポインタが返されてきたとしたら，オフセット部の加算だけでは 256(0x100) バイトの領域しか線形にアクセスできなくなってしまうのだ(いうまでもないことだが，どちらのポインタも 0x0000ff00 という同じ物理アドレスを指している)．それ以降の領域をアクセスしようとすると，0x0000：0xffff の次は 0x0000：0x0000 になってしまうので，とってもまずい(図 5-1 参照)．

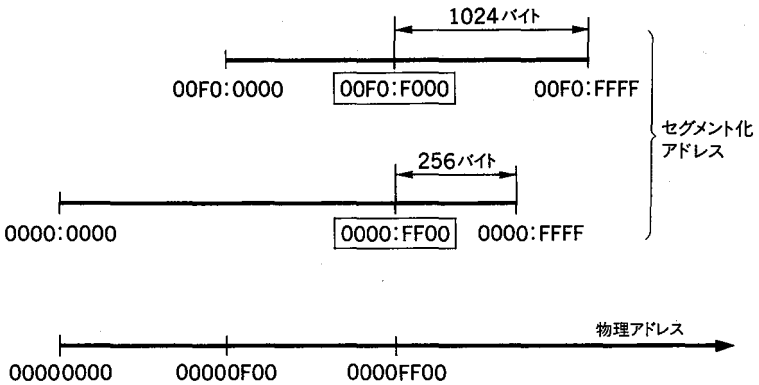


図 5-1　2 つの far ポインタ

とはいえ，C のライブラリ関数の_fmalloc()ではこの点についてちゃんと考えられていて，確保した領域がオフセット部だけで線形にアクセスできるようになっており，大変めでたい．しかし，上記の説明から明らかなように，同じ物理アドレスを示すからといってむやみやたらとポインタ値をいじって正規化したりすると，線形にアクセスできる保証がなくなってしまい，ポインタ比較すらできなくなってしまう．

プログラマとしてはこんなことにまで気を使わなければならないわけだが，だからといって

8086 ってなんて変な CPU なんだろう

などと嘆いてみても仕方がない．MS-DOS でプログラミングする以上，far ポインタを操作するときにはセグメントを十分に意識して，思わぬバグを作り出さないように気をつけるほかないのである．

malloc()とfree()の動作の秘密

まず結論から述べることにしよう.

malloc()した領域は,
free()した後もオペレーティングシステムに戻されない

たとえば自分でテキストエディタを書いたときのことを考えてみてくれたまえ. テキストエディタはテキストをメモリにロードして格納しておくために, たくさんのメモリを必要とする. だから, 頻繁にmalloc()を呼んで, メモリを確保する. そして, テキストの一部を削除したりしたときはそのテキストが格納されていたメモリ領域が不要になるので, その領域をfree()する. ところが実際には, このfree()した領域がオペレーティングシステムには戻されないのだ. つまり, オペレーティングシステムから見ると, テキストエディタはどんどんメモリを要求し続ける一方なのである.

そのため, MS-DOSでこのテキストエディタから子プロセス(COMMAND.COMとか)を起動しようとすると,「実行に必要なメモリが足りません」エラーが起こったりする. 当り前といえば当り前だ. なにしろテキストエディタはメモリを要求するばかりで, 返さないのだから.

このような何となく恩知らずというか仁義に反した(?)振舞いは, malloc()のストラテジ(戦略)に起因している. そもそも, UNIXのmalloc()は, メモリを割り当てるとき, 今のプロセス空間に十分なメモリがなければsbrk()という関数(MS-Cでは, brkctl()という内部ルーチン)によって新たなメモリブロックをプロセス空間に付け加える. sbrk()は, アドレスの大きい方に向ってプロセス空間を広げ, その先頭のアドレスを返す. sbrk()が広げたメモリ空間と, そのアドレス以下の空間は, プロセスが使ってもいいという意味になるのだ.

実際のところ, UNIXではsbrk()でメモリを確保していく一方でもあま

り問題ない．そこは仮想記憶のマジックでなんとかなってしまう．つまりプロセス空間の一部(あるいは全部でもよい)を物理メモリから「ひっぺがして」(ディスクに記憶しておいて)，他のプロセスにその空間を与えるのである．しかしMS-DOSは仮想記憶をサポートしていないので，プロセスが要求したメモリはそのまま物理メモリに「張り付いて」いる．そのため，その物理メモリを使いたくても使えずに，他のプロセス(子プロセス)は狭い思いをするか，実行できない羽目になるのだ．

ところが，MS-DOSのシステムコール体系は，sbrk()によるメモリ管理の体系とは異なっている．MS-DOS下で起動されたプロセスは，他のプロセス(親プロセス)の張り付いていない，最大のメモリブロックをプロセス空間として受け取り，必要な大きさを自分で判断して，その大きさに自分の空間を縮めるようになっているのだ．

注目すべきことに，MS-DOSはセグメント単位のメモリ管理をサポートしていて，実際にはメモリブロックの割り当て，解放(MS-DOSに戻される)が行える．だから，その気になれば，free()すればシステムにメモリが返されるようにできるはずなのだ．それなのに，MS-Cでもfree()した領域はシステムには戻されない．逆に言えば，MS-DOSの機能を無視してでも，sbrk()のストラテジをシミュレートしたかったのだろう．

実際にはMS-DOSでは複数のプログラムが並列に動くことはないので，MS-DOSのメモリ管理機構を使用しても，そんなにメリットはなかったのかもしれない．でもsbrk()はUNIXのシステムコールなのであって，Cのライブラリではないのだ．open()だってそうだ．そのシステムコールを隠すために，malloc()やfopen()があるのではないだろうか．Cのプログラムは UNIX(しかも特定のマシン上の)システムコールをガンガン呼んでいるのに，「移植性が高い」などと言うのは，正直に言わせてもらうと

ウソ八百，笑止千万

だ．

alloca()ってどう使う?

メモリを割り当てるのには,今述べたmalloc()がよく使われる.malloc()を使えばあらかじめ領域を確保しておかなくても済むし,プログラムの文脈に従って必要な量のメモリだけを動的に割り当てられるようになるからだ.しかし,malloc()による割り当てとfree()による解放を無節操に繰り返すと,主記憶がずたずたに分断されてしまって加速度的にmalloc()しにくくなる(遅くなる).

一方,fgets()で文字列を読み込むときなどは,malloc()なんか使わずに関数の内部でcharの配列を宣言して,そこに読み込ませるようにするのが一般的だろう.この手段を使えばメモリはスタックとして使用されるので,その関数が終了すれば解放されるし,スタックの伸び縮みに従ったメモリ管理なので主記憶をずたずたにすることもない.しかし,malloc()の場合とは逆に,あらかじめプログラマが決めた固定の大きさのメモリしか確保できないし,実際にそれを使わなかったときなども無駄に目をつぶらなければならないという難点がある.つまり,スタックに(配列)変数を宣言すると,その大きさはプログラムの生成時に決められる静的なものになってしまうのだ.しかし,なんとかスタック上に,しかもmalloc()のように動的にメモリ割り当てを行いたい.これを実現するのがalloca()なのだ.

alloca()の使い方はmalloc()とほとんど同じである.しかし,

確保した領域はヒープ領域ではなく,スタック上にある

という点がミソなのだ.そしてfree()する必要はない.関数から抜けたときに,自動的に,通常の局所変数と同じように解放されるからだ.

このことから分るように,alloca()は一時的なメモリ割り当てに使用されるものだ.しかし,そこに格納されたデータが別の場所とかで必要にな

ったら，その領域の大きさ分だけ malloc() したヒープ領域にコピーすればよい．

では，具体的な使用例をリスト 5-10 に示そう．このルーチンは，文字列を読み込んでそれを 2 バイトコードに変換するため，テンポラリの作業領域を alloca() で割り当てて変換作業を行い，それを別の関数がアクセスできるように，malloc() で割り当てた領域にコピーし，その領域へのポインタを返している．実際に 1 バイト文字を 2 バイト文字に変換する (2 バイト文字はそのまま) のは，translate() にやらせている．

```
/*
 * 特殊条件のチェックはやってないよ
 */
char *load_doublebyte()
{
        char buf[BUFSIZ];
        char *p, *q;

        fgets(buf, BUFSIZ, stdin);
        p = alloca(strlen(buf) * 2 + 1);
        translate(buf, p);
        q = malloc(strlen(p) + 1);
        return strcpy(q, p);
}
```

リスト 5-10　alloca() を使った領域確保

−Gt オプションはこう使う

さて，ライブラリの話題からは少し離れるが，メモリ管理の話題が続いたところで，ついでにデフォルトデータセグメントについての話をしておこう．

スモールモデルはデータセグメントが1つしかないからいいとして，ラージモデルのようにいくつものセグメントをもっているプログラムでは，プログラムが起動したときにDSとSSにアドレスされるセグメントとして，デフォルトデータセグメントというのがある．このデフォルトデータセグメントには，スタック，文字列などの「初期化された」静的データ，グローバル変数が格納される．つまりprintf()のフォーマット文字列などもデフォルトデータセグメントに入れられる．

ところが，UNIXマシン用のプログラムを移植しようとしたとき，このデフォルトデータセグメントに絡んで問題が発生する．移植しようとするプログラムがたくさんの文字列を含んでいるような場合，こうした文字列がデフォルトデータセグメント(64Kバイト)に入り切らないときがあるのだ．高度に複雑なプログラムであるnethack[*3]などはその良い例だろう．

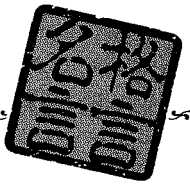
nethackをMS-Cでコンパイルすると，コンパイルの最中で，デフォルトデータセグメントが一杯になったというメッセージを表示してコンパイラが降参してしまう．nethackには，メッセージ類を収めた文字列が大量に含まれているので，こういう問題が起きるのである．これを解決する手段が，−Gtオプションなのだ．きっとあまり使ったことはないと思う．

−Gt32

と指定すると，32バイトを越えるようなデータオブジェクトはすべて，新しいデータセグメントに格納される．配列や文字列がこの長さを越えるなら，それらはみなデフォルトデータセグメントから追い出されるので，う

まくすればデフォルトデータセグメントが一杯になるという状況が回避できるのだ．そして，nethackの場合はそれがうまくいった．

−Gtオプションを指定しないと，−Gt32767を指定したのと同様の意味を持つ(−Gtを指定して，その値を省略すると−Gt256と同じ)．こんなに大きな値では，ほとんどの静的データはデフォルトデータセグメントに格納されることになってしまう．だから，ラージモデルでプログラムを書くのなら，−Gtオプションの使用を考えることも必要である．

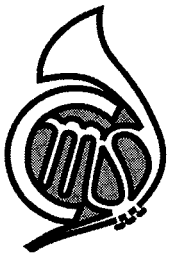


福崎、盆にかえらず

＜解説＞

プログラムや原稿などの納期が厳しいと，私生活を犠牲にしなければいけないこと．この本に関わった人間に限ったことではないだろう．

*3高度に複雑な，キャラクタ画面指向の打鍵練習プログラム．ハワイアンシャツは魔法をかけて，鎧の下に着るとよい．



ほんとに使うライブラリ関数はこれだけだ!

これまで，ほとんど入出力とメモリ関係の話にページを費やしてきたが，この2つの話題だけとってみても，ライブラリというのはなかなか奥が深いものだということが分っていただけたと思う．しかし，Cのライブラリにはこのほかにも死ぬほど多くの関数が揃っているし，どんな処理系でもライブラリ関数のマニュアルはたいてい分厚く，初心者ならいきなりめげてしまっても不思議はない．だが，ここで扱わなかった関数といっても，文字列関係の関数がやたら数が多いだけで別に難しいこともないし，初心者は別にして，まあ普通に使えばよいだけの話だ(とかゆう)．

大体，ライブラリ関数とかいって大きな顔して頭数だけは揃っているが，一生使わないような関数だってある．ひょっとしたら使うかも知れないから，一応マニュアルを一通り読んで，どんな関数があるか頭の隅に入れておけば支障はないのだ．

そこで，ライブラリの話題の締めくくりとして，数あるライブラリ関数の中で，実際に使う関数はどれかを紹介しよう．といっても，別に趣味とかで恣意的に選んだわけではない．『MS-DOS SOFTWARE TOOLS』シリーズのコマンドを開発した際に実際に使った関数，および，gcc，nethack，GNU emacsの各ソースで使われている関数を集計したのである．だから，かなり有意な結果が出たはずだ(?)．

集計結果は，**表5-3**から**表5-6**までに掲載してある．使われている関数全部を載せても意味がないので，ここではそれぞれ上位20番目までの関数を掲載した．

頻度	関数名	頻度	関数名
359	exit	85	write
298	fprintf	62	fclose
219	fputs	55	fopen
194	strlen	54	toupper
166	strcpy	54	close
128	intdos	48	atoi
112	printf	39	sprintf
101	malloc	36	read
92	strcmp	34	setmode
90	strcat	34	free

表 5-3 『MS-DOS SOFTWARE TOOLS』1〜3 で使われている関数一覧

頻度	関数名	頻度	関数名
441	fprintf	31	strcat
342	printf	28	fflush
264	abort	24	ungetc
153	alloca	23	fwrite
88	strcmp	20	sprintf
81	strlen	20	fopen
45	strcpy	18	malloc
41	fputs	17	strncmp
39	exit	13	realloc
33	free	12	fputc

表 5-4 gcc で使われている関数一覧

頻度	関数名	頻度	関数名
302	printf	51	exit
216	strncmp	47	unlink
168	strcmp	47	fclose
160	strlen	46	strcpy
130	free	46	fopen
86	write	44	signal
72	strncpy	43	yn*4
60	open	36	abs
59	fflush	33	puts
55	close	33	getenv

表 5-5　nethack で使われている関数一覧

頻度	関数名	頻度	関数名
117	signal	52	exit
94	strlen	49	write
94	strcmp	46	open
91	fprintf	44	read
73	strcpy	43	malloc
71	close	35	sprintf
66	free	34	lseek
60	alloca	31	strcat
54	printf	26	fflush
54	abort	21	getpid

表 5-6　emacs で使われている関数一覧

TOOLS で exit()が堂々一位に輝いているのは，山田君(仮名)が自分のプログラムの中でやたらと呼んでいるためだ．intdos()なんていうのが上位にきているところなど，さすがに MS-DOS 上のツールだという特徴が表れている．gcc はコンパイラであるだけに，fprintf()などの出力関数や文字列処理関数が上位を占めている．alloca()がかなり上位にきている点も

*4ここで登場する関数 yn の趣を理解できる人はかなりの通だ．

目を引くところだ. nethack は対話型のプログラム(ゲーム)であり,メッセージの出力が多いため, printf()や文字列処理関数が使われる頻度が高い. 高機能エディタである emacs の場合にもやはり文字列処理関数が上位にきているが, 対話処理でシグナルを処理する必要上からか, 1位の座は signal()が獲得した.

ただ誤解しないでほしいのは, ここでの結果はソースコードでの出現頻度であって, 実際にその関数が呼び出されている頻度ではないということだ. だから, この表に載っていない関数でも, プログラムの実行中に頻繁に呼び出されている可能性はある. だが, この表からでも, プログラムの目的によって使われる関数にそれぞれ特徴があるのがわかるはずだ.

読者の方々も, この集計結果を参考にしてライブラリ関数を使うようにすれば, きっと

いかにもプログラミングの達人が作ったようなプログラム

が書けるようになるはずだ!(知らないけど)

もっと速く，もっと小さく

「努力ゥ？　エエ～バカみたい，ダッサ～い」というのが1980年代のノリだった．しかし，努力が罪悪のように言われたクルクルパ～な時代を，燃えさかる情熱と鉄のような意志によって生き延びた人々がいる．『MS-DOS SOFTWARE TOOLS』を開発したプログラマたちだ．

彼らは，心ならずとも(？)リリースした「普通のでき」のプログラムでは決して満足せず，プログラムの高速化とシェイプアップを目的に，黙々と(あるいは絶叫と共に)自らの作品の改良に青春を捧げてたのである(したがって，こうした改良の結果は，ユーザーへは反映されていない)．

この章は，TOOLSプログラマの一人(あまり能力は高くないが一本人談)がCウィザードを目指す読者のために書きなぐった，フィルタ系プログラムのチューニング講座である．彼らTOOLSプログラマ達が，古いマンションの一室でどんな作業に労力を費やしていたかを垣間見せてくれるだろう．

あるTOOLSプログラマのこだわり

酒の飲み方やスピーカーの置き方など，この世にはいろいろとこだわる人がいるが，そうした人は，その分野に興味のない人の目には単なる変人にしか映らないようだ．プログラマのこだわりも，一般人から見れば理解できない部分があるかも知れない．確かに，「C言語の常識と非常識」で述べたようなことばかりにこだわるプログラマはどうかと思う．だが，本物のプログラマならば，

プログラムの完成度をとことんまで高めたい

という何にも優る信念を持っているはずだ．我々もCウィザードを目指す限りは，たとえどんなに眠くても，

一応動くからこの辺でもういいや

などという安易な妥協をしてはならない．

プログラムの完成度を高めるためには，考えなくてはならないことが山のようにある．何を取り，何を諦めるか，チューンアップ作業のポイントの1つは，その取捨選択にあるといえる．

プログラム３原則

プログラムの良し悪しの前には，プログラムが完成しているかどうか，という段階がある．実は，世の中に出回っているプログラムには，このレベルをクリアしていないものがけっこう多い．致命的なバグが残っていたり，ドキュメント(マニュアル)の不備で機能がさっぱり理解できなかったり，という類のものだ．「誰でも使えるようになる」かどうかは商品としての完成度の問題だが，「その分野に詳しい人間」にも使い方が分からないようでは，プログラム自身の完成度が低いと言わざるを得ない．

SFでは，アシモフが提唱した「ロボット３原則」なる規則が有名だが，これに倣って「プログラム３原則」でも定義してみよう．

〔第１条〕
プログラムは，データを破壊してはならない．

プログラムが暴走し，システムがハングアップしても，データさえ無事ならば何とかなる．しかし，データ(ディスク上のファイルなど)を壊されると，ユーザーは泣くに泣けない思いをすることになるだろう．ユーザーの操作ミスならいざしらず，プログラマの不注意でデータを壊さないようにしたいものだ．一見正常に動作しているプログラムでも，極端なデータを処理させたときに，ボロが出ることも多い．

〔第２条〕
第１条に反しない限り，
プログラムは止まらなくてはならない．

暴走族は傍迷惑な存在だが，プログラムの暴走も同じ程度に許せない．どんなに便利なプログラムでも，暴走するプログラムでは使ってもらえま

い．ここで「止まる」と言っているのは，ハングアップすることではなく，終了すること，つまりDOSの場合は終了のファンクションコールを発行することだ．エラーチェックを厳しくし，エラーのときにもきちんと止まるプログラムを作って欲しい．もちろん，暴走させることでデータを破壊しないで済むのであればこの限りではないが，ロボットと違い，プログラムではそんな場合はまずないだろう．

〔第3条〕
第1条と第2条に反しない限り，
プログラムはユーザーの指定どおりに動作しなくてはならない．

これまた当然のことのはずだが，複雑なオプション指定をするとエラーになったり，挙動がおかしくなるプログラムが時々あるのは困ったものだ．

ここに挙げた3原則は，どれも当たり前のことだ．プログラムをチューンする前に，読者のプログラムがこれらをきちんと満たしているか，もう1度確かめて欲しい．

「でき」を決める3つの柱

さて，プログラムが3原則に合格したとしても，それだけでは使い勝手のよいものとは限らない．たとえば処理速度1つとっても，プログラムの書き方次第で何十倍もの差が簡単についてしまう．

仕様についても，プログラマとユーザーの価値観によって，好みは大きく変わるだろう．たとえば私の周辺では，「オプション文字では大文字と小文字は区別する」というのが常識になっている．MS-DOSは，UNIXシステムを猿真似している点が多いので，コマンドの仕様についても，UNIXの流儀を借りているのだ．

ところが，TOOLSのユーザーの中には，「大文字と小文字を区別するのはけしからん」と言う方もいる．普段から仮名換FEPを使用しているユーザーの中には，CAPSロックをかけっぱなしにしている人もいる．このような人にとって，TOOLSの仕様は使いにくいものに違いない．これはもう価値観の闘いで，一歩間違えると宗教戦争にもなりかねない問題である．

このように，オプションの指定方法などについては人それぞれの意見があるものだが，たいていの人が賛同するスペックというのも存在する．

●処理速度

ずばり，速ければ速いほどいい．これが当てはまらないのは，リアルタイムゲームなどのみである．

●保守性

ユーザーにはまったく関係のないことながら，保守性も高ければ高いほどいい．保守性の高いプログラムは，書替えが容易なため，仕様の整理や速度向上がしやすい．また，バグが発見されたときも，その対応の早さが期待される．

●プログラムの都合による制限

緩ければ緩いほどいい．ここで言っているのは，「処理できるファイルの最大サイズ」や「1行の最大長」などといった制限のことだ．

この他にも，プログラムのサイズなども重要視されやすい項目ではあるが，ハードウェアの進歩が解決してくれることもあるので，さまざまな要因をにらみながら，チューンアップのポイントをどこに置くかを決めなくてはならない．

高速化との果てしない戦い

プログラムの高速化を行う際には，「どこを高速化するのか」をはっきりさせておかないと，手間の割に効果が上がらないことになってしまう．プログラムのどの部分(関数，ブロックなどの単位)で処理時間を食われているかを予め計測し，その部分に労力を集中させるのが高速化の原則だ．

UNIX ではプログラムのどの部分が処理時間を消費しているのかを測定するツールが用意されているが，MS-DOS では機種依存のツールがいくつか販売されているだけで，あまり普及しているとは言いがたい．テストプログラムを作ってみたり，プログラムの一部を書き替えて，速度がどの程度変化するかで，問題の部分を絞り込んでいくしかないだろう．

たとえば，リスト 6-1 のプログラムの実行時間を測ってみれば，フィルタ系のプログラムで入出力のオーバーヘッドがどの程度を占めるか，大体の見当がつくはずだ．

リスト 6-1　入出力のオーバーヘッドを計測するプログラム

```
#include <sys/types.h>
#include <sys/timeb.h>
#include <fcntl.h>
#include <stdio.h>

char    *buf;
int     size;

double  diff_time(a, b)
struct  timeb   *a;
struct  timeb   *b;
{
        double  res;
```

```
        res = a->time - b->time;
        res += ((double)(a->millitm) - (double)(b->millitm)) / 1000.0;
        return res;
}

doit()
{
        register int    cnt;

        while((cnt = read(0, buf, size)) > 0)
                write(1, buf, cnt);
}

main(argc, argv)
int     argc;
char    **argv;
{
        char    *malloc();
        struct timeb    start_time;
        struct timeb    end_time;

        setmode(0, O_BINARY);
        setmode(1, O_BINARY);
        if ((size = atoi(argv[1])) < 1) {
                fprintf(stderr, "%s: invalid value\n", argv[1]);
                exit(1);
        }
        if ((buf = malloc(size)) == NULL) {
                perror(NULL);
                exit(1);
        }
        ftime(&start_time);
        doit();
        ftime(&end_time);
        fprintf(stderr, "buffer size = %5d bytes\ntime = %f sec.\n",
            size, diff_time(&end_time, &start_time));
        exit(0);
}
```

さて，チューンする箇所を絞り込めたら，次に，「どう高速化するのか」を考える．一般的に，速度向上に効くのは，アルゴリズム，入出力，そしてループの最適化であろう．この3点を押さえるだけで，プログラムの性能はかなり向上するはずだ．もちろん，プログラムの性格によっては，より特殊な最適化が非常に有効なこともある．仕事仲間や友人と，あれこれ議論を戦わせてみるとよいと思う．

A+D=P

アルゴリズム，といって，誰もが思い付くのはソートのアルゴリズムではないだろうか．アルゴリズムのバイブル『A+D=P』*1などを読むとよく分かるが，同じ処理をするにしても，アルゴリズムの選び方次第で処理速度は大幅に違ってくる．プログラムがどのようなデータを扱うのか，データの量はどの程度なのか，といった見積りをきちんと行って，それにふさわしいアルゴリズムを選択したい．

たとえば，ソートアルゴリズムの代表選手クイックソートにも，他のアルゴリズムに劣る場合がいくつかある．1つのアルゴリズムに頼りきることなく，さまざまな方法を客観的に比べる能力が必要といえよう．

もう1つ，「使われ方を考慮した高速化」という手法がある．機能の豊富なコマンドでも，実際に利用されるのはごく一部だけのことが多い．たとえば，UNIX の fgrep では複数パターンの検索機能を持っているが，たいていの場合，ユーザーはコマンドラインから1パターンだけ指定する，という使い方をしている．言い換えると，ほとんどのユーザーは「複数パターンの同時検索」機能を使っていないということだ．

このような場合に，パターン数が1の場合を特殊扱いして，別のアルゴリズムを適用すると，一般的な使い方をする場合の実行速度を向上させることができる．同様の例として，シェアウェアの pkarc のように，複数のアルゴリズムの中から最適のものによってファイルを圧縮するものもある．万能のアルゴリズムが存在しない問題には，こういったアプローチも有効といえるだろう．

また，アルゴリズムに関連して重視しなくてはならないのが，データ構造だ．これをきちんと設計できるかどうかで，速度，メモリの消費量，プログラムの拡張性などに，大きな差がついてしまう．例として，sortの処

*1N.Wirth 著　片山卓也訳 『アルゴリズム＋データ構造＝プログラム』(日本コンピュータ協会刊)

理を考えてみよう．入力データ(テキスト)をメモリに保存するときに，テキストのまま保存するのか，内部構造(例えば，フィールド単位に分解して)に変換するのか．行単位でリンクリストを使うか，巨大なバッファにべた詰めにし，行の先頭を指すポインタ配列を用意するのか．内部のデータ表現がテキストのままでないとしたら，テンポラリファイルには内部構造のまま書き出すのか，テキストに再変換するのか．

こんな具合に，たかがフィルタでも数多くのデータ構造を検討する必要があるし，それがプログラムの使い勝手を左右することになる．ゆめゆめ馬鹿にすることなく，きちんと検討したいものだ．

入出力の隘路

フィルタ系のプログラムでは，データの入出力(read()/write()のようなライブラリ関数よりも下位の部分)の占めるオーバーヘッドがけっこう多い．オーバーヘッドの割合は処理によるが，wcやfgrepのように単純なものだと，処理時間の半分以上が入出力に食われていることが多い(自作のプログラムによる測定結果)．

こんなプログラムだと，いくら内部のルーチンを高速化しても，性能向上の程度はたかがしれている．たとえば，入出力で処理時間の75％を消費しているプログラムでは，内部ルーチンを10倍高速にしても，性能は30％程度しか向上しないが，入出力を2倍高速にするだけで，性能は60％向上する．

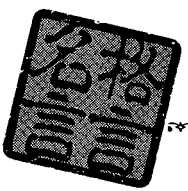
もっとも，入出力を2倍高速にする，というのは，簡単なようでなかなか難しい．これを実現するためには，入出力がどのような経路で行われているかをきちんと押さえ，その経路を簡単にすること，そして入出力の回数を減らすこと，この両面から攻める必要がある．

たとえば，Cのプログラマにとって非常に馴染み深いライブラリ関数printf()．たいていのプログラマはこれを気軽に使う．だが，速度の面から見ると，printf()は非常に低速な関数で，チューンの際には真っ先に粛清の

対象にもなる.

printf()を呼び出すと，内部ではかなり複雑な経路を通ってから実際の出力がなされる．図6-1は，printf()の内部で，どのような関数が順に呼び出されているかの概念図だ(実際には，使用されている関数名は多少違うことが多い．その辺はバージョンにも依存するので，ライブラリを解析してみて欲しい)．

もし，プログラマがMS-DOSのファンクションコールを直接呼び出せば，3段階分の中間経路を省略することができ，相当速度が向上する．その代わり，printf()の持つ複雑な書式制御機能なども，すべてアプリケーション内部で実現する必要があり，保守性も相当低下するので，性能とのバランスを取りながら，最適な選択をしてみて欲しい．



50個100個

＜解説＞

ローカル変数をひとつの関数内で大量に使うと，ローカル変数にアクセスするのにMS-Cの場合には8ビットのオフセットでは間に合わなくなり，16ビットのオフセットでアクセスするコードが生成されるために，プログラムの実行が遅くなってしまう．それがいやな場合には，たとえ100個の変数を50個に減らしても50歩100歩，どうせならアルゴリズムや関数の構造から設計し直すのが一番かもしれない．

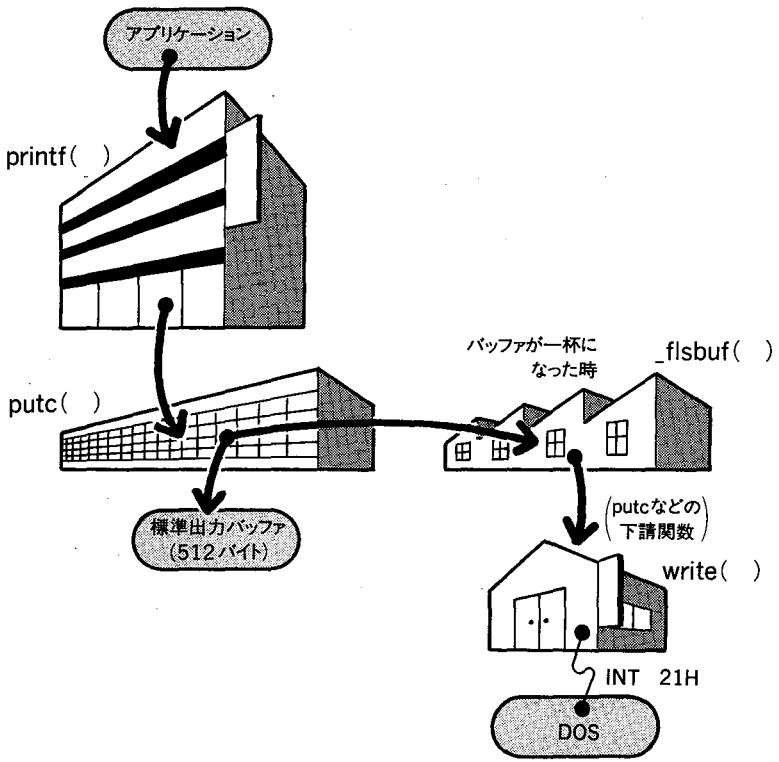


図 6-1　printf()から INT 21H まで

バッファリングでまとめ買い

入出力を高速化するためのもう1つの方法は，

入出力回数を減らす

ということだ．実社会でも，まとめ買いをすると安くなるものは多い．これはプログラムの入出力でも同じで，一度にまとめて入出力すれば，それだけDOSやライブラリのオーバーヘッドが小さくなる．

これを確かめるためのプログラムがリスト6-1だが，同じバイト数のデータを出力する場合でも，1バイトずつ，ちまちまやるのと，16Kバイト単位でやるのとでは，120倍以上の差がついている．表6-1は，バッファサイズを変化させながら，処理速度を計測した結果だ．

バッファサイズ（バイト）	1	16	256	512	1024	2048	4096	8192	16384
実行時間	357.02	24.17	4.12	3.68	3.18	2.96	2.80	2.76	2.74

表6-1　バッファリング効果

プログラムの内部処理にもよるが，速度優先で作成するのであれば，入出力とも十分バッファリングし，ライブラリやDOSのファンクションコールの呼び出し回数を最低限に抑えたい．ちなみにDOSのファンクションコールでは，1度のコールで最大65535バイトまで扱える．

もっとも，バッファリングによって使い勝手が悪くなることもある．たとえばgrepのようなツールでは，数Mバイトのファイルを検索し，結果はたったの100バイト，ということも珍しくない．このような場合に，出力デバイスがファイルの場合はバッファリングを行っても問題はないが，画面やプリンタの場合，全ファイルの検索が終わるまで出力を見られず，

ユーザーがいらいらするかもしれない．

つまり，一口に速度向上と言っても，プログラムの起動から終了までの速度だけでなく，ユーザーにとっての起動から終了(結果を得る)までの速度も考慮しなくてはならないのである．「手に馴染む」道具作りのためには，人間工学(でいいのかな？)の基礎知識もおろそかにはできないのだ．

ループを最適化する

プログラムの最適化では，非常に有名な経験則がある．それは，

ソースコードの 10 %の部分が，
処理時間の 90 %を消費している

というもので，「10-90 ルール」などと呼ばれている．これが正しいとすれば，プログラムの 10 %の部分を 2 倍速くするだけで，全体の処理時間は 55 %に縮まることになる．

大変なのは，どこが問題の 10 %なのかを見極めることだが，一般には，ループの内部を疑うのが常識になっている．特に，ループが 2 重 3 重になっている部分では，その内側のループを最適化するだけで，びっくりするほど高速になるものだ．

リスト 6-2 と**リスト 6-3** は，まったく同じことをする関数だが，実行速度は 140 倍近く違う．これは極端な場合かも知れないが，ループの中は，最適化のしがいがある，ということを頭に入れておくと，チューンアップ作業もスムースにいくだろう．

```
main()
{
        int     i;
        int     j;
        int     k;
        long    l;

        l = 0;
        for (i = 0; i < 100; i++)
                for (j = 0; j < 100; j++)
                        for (k = 0; k < 1000; k++)
                                l++;
        printf("%ld", l);
}
```

リスト6-2　3重のループ

```
main()
{
        register int    i;
        register int    j;
        long    l;

        l = 0;
        for (i = 0; i < 100; i++)
                for (j = 0; j < 100; j++)
                        l += 1000;
        printf("%ld", l);
}
```

リスト6-3　リスト6-2を最適化したループ

保守性のために保守をする

プログラムの保守性については，これまでにさまざまな人たちが研究し，それなりの成果を上げているが，その中でももっとも知られているのが，ダイクストラらの提唱による，

構造化プログラミング

であろう．ダイクストラの名前は知らなくても，「構造化」という単語にはしばしばお目にかかっているに違いない．

C言語は，この影響を受けた言語で，一応構造化プログラミングができる仕様を持っている．一応，と断ったのは，Cの言語仕様は全般的に制約が緩く，プログラマがその気になれば，簡単に非構造的プログラムを作れるからなのは，ここまで読み進めて来た読者には言うまでもないことだろう．

構造化プログラミングも，プログラムの保守性を上げるための重要な手段だ．しかし，私の主観からすると，これだけではプログラムの保守性にも限界があるように思える．そこで，プログラムがよほど単純な場合を除けば，プログラムの

「虫干し」

が，保守性に大きな影響を与えると主張したい．

虫干しといえば，しまっておいた本や着物を梅雨明けに陽に当てて殺菌することだ．プログラミングの虫干しも同様で，できあがったプログラムをそのままにしておかず，時々(半年～1年に1度位)引っ張り出して改良に挑むとよい．もちろん，そんな作業をするのは天気のいい日に限る(ウソ)．

プログラムも，作ってからある程度期間を置くと，当時は気が付かなかったバグがみつかったり，時代の変化に対応させたくなることも多い．プ

ログラマ自身も成長しているので，より効率的でスマートなプログラムが作れるようになっていることだろう．コンパイラのバージョンアップに伴い，プログラムの書替えが必要になっているかもしれない．

一度できあがったプログラムをバージョンアップせずにとことん使い続け，寿命が尽きたら最初から作り直す，というのもやりようの1つではあるが，たいていのコマンドでは適宜改良していく方が，ユーザーにとってのメリットは大きいような気がする．ある程度期間を置きながらプログラムのデバッグ，バージョンアップをやるようにすれば，プログラムを時代に合わせて成長させられるし，第三者的な視点からプログラムをチェックし，より保守性の高いものに改良できるかもしれない．つまり，継続的に保守を続けること自体が，プログラムの保守性にとっては大切なことなのだ．

だが，この方法にも問題が1つある．長期間放って置いたプログラムを眺めても，何をやっていたのか，関数の名前と機能はどうなっていたか，といった，細かいことをほとんど忘れていることだ．確かに，プログラムを客観的にチェックするには理想的な状況だが，バージョンアップやバグfixのためにソースを引っ張り出したのに，その仕様を忘れているのでは話にならない．

大型計算機のプログラマの場合，プログラミングの前に，製品仕様書やらモジュール定義書，あるいは各モジュールをテストする条件を定めたテスト仕様書といったドキュメントを，山のように用意するという．こうしないと，大規模なプログラムを多人数のメンバーで開発できないのであろう．

アマチュアプログラマの場合，プロと違って毎日プログラミングを行っているわけではない．当然，自作したプログラムでも，完成したあとは徐々にいじらなくなり，その中身(プログラムの構造など)の記憶も薄れていく．

プログラムを改良したり，バグを取ろうと思っても，そのプログラムの構造を忘れているのでは仕事にならない．結局，プログラムを最初から読み直し，何をやっているかを解読する必要がある．こんな無駄を避けるた

めにこそ，ドキュメントをきちんと作っておくのだが，趣味のプログラマに，使うかどうかもわからない，無味乾燥なドキュメントを書けというのも無理な話だろう．

そこでお勧めしたいのが，

「プログラム日記」をつける

ことだ．新しいプログラムを作るごとに，〝diary〟とかいうファイルを用意し，プログラムをいじったときには，何をやったかをこまめに記入していく．プログラムのコメントやアップデートログは英語(もどき)で書くことが多いが，プログラム日記は気楽に書ける日本語(というか，母国語)で書く方がいいだろう．日記には，どんな作業(試行錯誤)をしたかや，ベンチマークの結果を記録しておく．

さて，こんな日記をつけて何の役に立つのか，というと，プログラマの記憶を掘り返すのによく効くのだ．「ベンチマークで，TOOLS の fgrep より 3 倍高速な値を記録」とか，「クイックソートにしても速度に大差なし．サイズは 500 バイト増加．元に戻す」，「16bit　FAT/80MHDD のとき，FAT を壊すバグを発見．冷や汗を流しながら修正」等といった文章を読み返すうちに，忘れたと思っていた感情が，鮮やかに蘇ってくるのが分かるだろう．一生懸命作ったプログラムならば，忘れたつもりでもけっこう覚えているものだ．「そういえば，あの頃は『〝{〟の位置論争』に熱中していたな～」なんてつぶやきながら，当時の状況をかなり思い出せるに違いない．回顧趣味のある人なら(実は，私もその 1 人である)，暇なときに読んでは，

「昔は若かったなー」

などと苦笑いできるだろう．図 6-2 は，TOOL-3 の MAKE コマンドの日記の一部だ．他人から見ると馬鹿げた文章が並んでいるが，プログラム開

発に関係した人間にとっては，よき思い出になることは保証する．

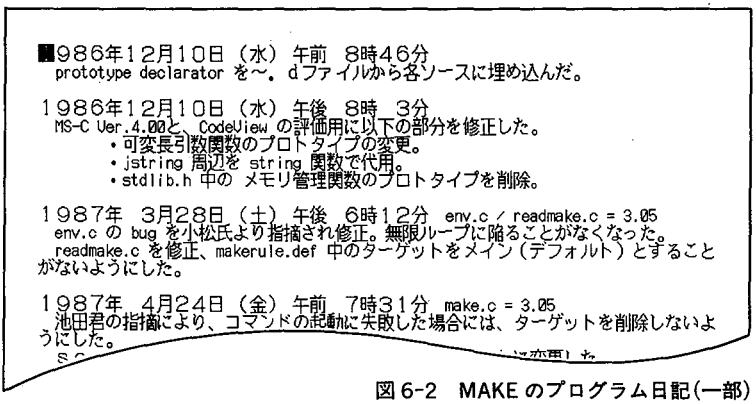


図 6-2 MAKE のプログラム日記(一部)

つまり，プログラム日記は，プログラムへの感情移入を助ける手段なのだ．ソフトウェアハウスなどでは，プログラムの作成者と保守担当が違うことも珍しくないので，こんな日記では何の意味もないだろうが，アマチュアならば，アマチュアなりの管理方法を取るのも一興ではないだろうか．

正しい仕様をめぐる闘い

仕様といって，読者は何を連想するだろうか．オプション，ファイルの指定方法，画面設計，人それぞれで色々なものが挙げられると思う．

コンピュータ文化には，仕様の何たるかを端的に表した格言がある．いわく，

ドキュメントに書かれれば，バグも仕様に変わる

つまり，仕様というのはドキュメントに記載されたプログラムの性格，ということができる．

だが，実際には，ドキュメントに記載されるプログラムの仕様なんて，氷山の一角に過ぎない．実際のプログラムの使い勝手は，ドキュメントに現れない仕様を，プログラマがどこまで煮詰めているかで決まるものだ．

ドキュメントに記載された仕様を「表の仕様」とすれば，ドキュメントに記載されない仕様は「裏の仕様」ということになるだろうか．たとえば，エラーメッセージ一覧が表の仕様だとすれば，いかにエラーの発生を抑えるか，が裏の仕様だ．オプション一覧が表なら，オプションの自由な組み合わせを確実に保証するのが裏である．

ドキュメントに現れにくい仕様の中でも，主なものをいくつか挙げてみよう．

● invalid(非正規な)データの扱い

(例)テキストを処理するコマンドにバイナリデータを食わせた場合の対応

●スタックのオーバーフロー防止
(例)環境変数のサイズが大きいときに，スタックのオーバーフローが出るのを防止する
●最適なエラーメッセージ
(例)コマンド名をリネームされたときに，エラーメッセージも変更する
● exit code(リターンコード)の保証
(例)エラーの有無によって，必ず0，非0を返す

こんなところに気を配っても，9割以上のユーザーは気がつかないだろうが，「何となく使いやすい」と評価されるに違いない．評価されるプログラムというのは，たいてい，こういった配慮がきちんとなされているものなのだ．しっかりとしたプログラムを書けるプログラマでなければ，こういった瑣末なことにまで注意を払う余裕はないに違いない．

仕様というものは，曖昧模糊とした価値観によって決められるものだから，価値観の違いによって同じ仕様でも評価が真っ二つに分かれるものだ．特にそれが裏の仕様となると，「できが悪い」とみるか「かゆいところに手が届く」とみるか，好き嫌いがはっきりしやすいようだ．

一見不毛な闘い，宗教論争になりそうなテーマではあり，実際そうなることも少なくない．そんなときは，

中島みゆきの「世情」

でも口ずさみながら，自分が正しいと思う道を選ぶしかないだろう．

冗談はさておき，冷静さを保てるのであれば，こういった問題の議論，考察を推し進めるのもいいことだと思う．自分が，どのような論拠でその仕様を選んだのか，その仕様の問題点はどこにあり，解決するにはどうすればいいのか，知恵を絞ってみて欲しい．

こんな問題の一例として，私らTOOLSの開発メンバーが直面した，テキストファイルの問題を紹介しよう．

テキストファイルって何だっけ?

DOS上でのプログラム開発に取り組んでいるプログラマであれば，「テキストファイル」と言われれば，何のことだかすぐに分かると思う．要は，TYPEコマンドによって画面に表示させることのできるファイルのことだ．一見，分かりきった概念のはずが，TOOLS開発では，その定義をめぐって，議論が沸騰した．

「テキスト」と「バイナリ」という概念は，MS-Cだけではなく，MS-DOSのレベルにも存在する．たとえば，COPYコマンド(DOSの標準コマンド)では，オプションによってファイルをどちらのモードで扱うかを指定できる．しかし，MS-DOSのマニュアルには，テキストファイルの定義が一切ない．したがって，現在のテキストファイルの仕様は，プログラマごとに違っているのが実情だ．

大きな問題は2つある．1つはファイルターミネータの定義，もう1つはラインターミネータの定義だ．

定義されないファイルターミネータ

一般にDOSのファイルターミネータは'^Z'，つまり01AHとされている．実際，DOSの標準エディタEDLINは，ファイルの最後に^Zを付加する．TYPEコマンドも，^Zで処理を終了する．

だが^Zは，ファイルサイズをクラスタ単位でしか管理していなかったCP/M時代の遺物で，ファイルサイズがバイト単位で管理されているDOSでは不要なもののはずだ．リダイレクト〝>>〟を使う場合にも，^Zの存在は邪魔であり，現時点では^Zはなくした方がよいと考えられる．だいたい，

```
echo hello > foo
```

としたときに，fooの最後に^Zが付かないことからしても，^Zはオプショナルなもので，なくすべきではないだろうか．

つまり，^Zをどう扱うかが問題になるのだ．選択肢はいくつかある．

1. ^Zを完全に無視し，入力データ中の^Zは削除(無視)してしまう
2. 一般の文字として扱う
3. 入力データ中の^Zはファイルターミネータとして扱うが，一切出力しない
4. 入力データ中の^Zをファイルターミネータとして扱い，出力データの最後にも^Zを出力する

結果的に，TOOLSのメンバーが選択したのは3.の方法だ(図6-3)．データの透過性という観点からは，2.の方法が最も優れている(余計な処理をしない，という点で)のだが，^Zを世の中からなくすためには，1.か3.の方が望ましいし，現状でTYPEやEDLINなどのコマンドが^Zを認識する以上，入力データ中の^Zはファイルターミネータとして扱うべきだ，というのがその理由だ．COMMAND.COMのリダイレクト(">>")を使う上でも，3.の方法が一番無難である．また，MS-Cのテキストモードでも^Zは3.の方法で扱われており，その点でも開発が容易になる．

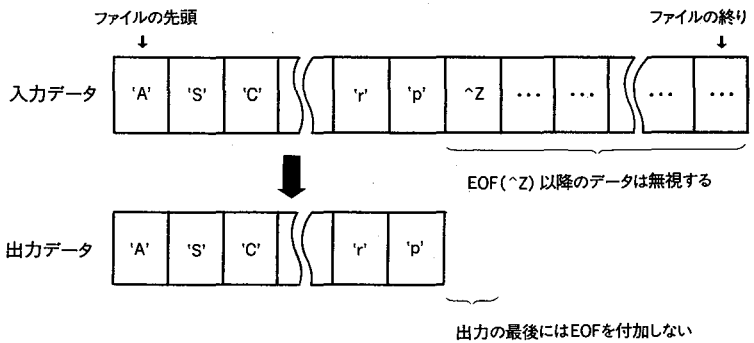


図6-3　TOOLSにおけるファイルターミネータの扱い

TOOLSにおけるファイルターミネータの扱いは，一般の基準ではそこそこまともなものだと思うが，これを悪用すると簡単に〝邪悪(Evil)〟な仕様のプログラムを作ることができる．

TOOLSには，オンラインドキュメント機能をサポートするためにmanコマンドが用意されている．ドキュメントファイル(man.doc)は，大雑把に言うと，図6-4のようなフォーマットで格納されている．

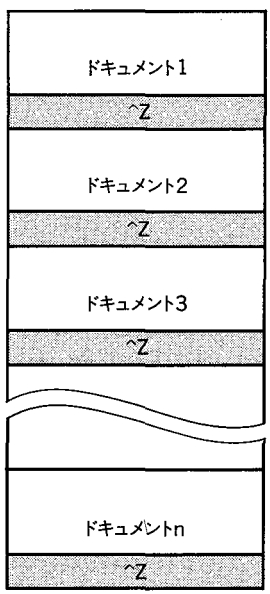


図6-4　man.docのフォーマット

manコマンドが起動されると，man.docをオープンし，指定されたドキュメントの先頭にシークする．次にman.docのファイルハンドルを標準入力(ハンドル0)にリダイレクトし，その状態でページャ(moreなど)を起動する．起動されたページャは標準入力から順に読み出し，EOFコードを読み出したところで処理を終える．

というのが，設計の基本思想だった．こうすれば，指定されたドキュメントを man.doc から抜き出し，テンポラリファイルに格納する必要がなくなるので，処理がかなり高速化されるのである．

ところが，この仕様はユーザーからさんざんな不興を買った．PDS として流れている less(more よりも強力なページャ)では，EOF コードを一般文字として扱っているため，指定したドキュメント末の EOF コードを超えて，次のドキュメントまで出力されてしまったからである．

考えてみると，「EOF コードは時代遅れだ」と言いながら，その EOF コードを利用したプログラム設計をしているわけだから，随分と無責任な話だ．

実は，この仕様を提案をしたのは私である(man コマンドを作成，改良したのは別人)．現在では

深く反省

しているので，どうか許してやって下さい．

ラインターミネータの悩み

ラインターミネータについても，正しい仕様が決まっていない(DOS のドキュメントに記載されていない)のが問題になった．一般に，DOS のテキストファイルでは，〝CR/LF〟というバイト列によって，ラインターミネータを表す．'CR'，もしくは'LF'単独では，表示などがおかしくなることが多い．

C 言語では，ラインターミネータを'¥n'，つまり LF(DOS の場合)一文字で表現している．TOOLS のコマンドはほとんど MS-C で記述されているが，どちらの流儀に従うべきなのだろうか．また，入力データ中に CR や LF が単独で出現した場合には，どう扱うのがいいのだろうか．

例によって，DOSの標準コマンドを調べてみた．行の概念があるコマンドということで，EDLINとFINDを調べてみたところ，どちらもLF1文字をラインターミネータとして扱い，CRは一般の文字と同じ扱いになっていた．

TOOLSでは，これらのコマンドとも違うアプローチを取った(図6-5)．

入力データ　ラインターミネータはLFの1バイト
LFの直前に表れたCRは無視(削除)する
単独で表れたCRは一般文字として扱う

出力データ　ラインターミネータはCR/LFのバイト列に変換
CRはそのまま出力

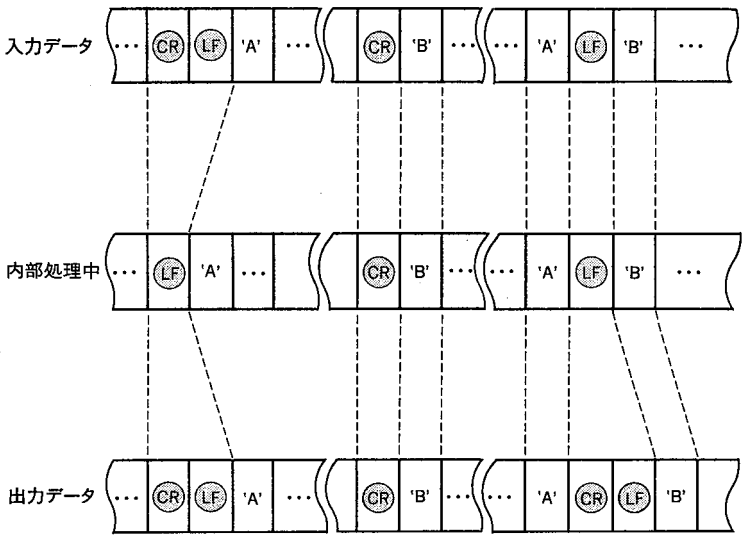


図6-5　TOOLSにおけるラインターミネータの扱い

実を言えば，これもMS-Cのライブラリの仕様をそのまま採用したものだ．これは，MS-Cのライブラリがそのまま使える，という点もさることながら，MS-DOS環境でテキストを処理する場合には，この仕様が一番使いやすいように感じられたからでもある．たとえば，入力データ中ではLF1文字でラインターミネータとすることによって，UNIXのテキストファイルをそのまま処理し，結果をDOSのテキストファイルに変換できる．

この仕様は，多くのユーザーにとっては気に入るもの，あるいはどうでもよいものだったようだが，一部のユーザーにとっては余計なおせっかいだったようだ．「テキストファイルとバイナリファイル，という区別が気に入らない」という御意見も頂いた．私個人は，DOSの環境を利用する限り，TOOLSの仕様が適当だと思っている．テキストファイルを特別扱いしないためには，DOS本体とDOSの文化そのものを変革する必要があるだろうし，現状ではそのような仕様のコマンドは，実用性に乏しいように感じられる．

8あれば9あり

＜解説＞

charのサイズは8ビットだとは限らない．charが9ビットのマシンだってあるのだ．このように，特定マシンでのデータ型のサイズに寄りかかったプログラムを書いて楽をすると，あとで移植するときに苦労するという先人の教訓．

やっぱり気になるスリーサイズ

2，3年程前から「もう，プログラムサイズを気にする時代ではない」といった論調の意見をよく見かけるようになった．確かに，10年前と比べれば，メモリや補助記憶(ディスク．当時は紙テープやカセットテープ)も圧倒的に大容量になり，同じレベルで考える必要はなくなっている．しかし，プログラムサイズの小ささは，速度や使い勝手とも密接な関係にあり，

可能な限り小さくすべきだ

というのが私の意見だ．

たとえば，最近はやりのノート型/ブック型パーコンにしても，デスクトップ・システムと比べて補助記憶の容量が小さく，プログラムサイズが小さくないと使い勝手に問題が起きる．大容量HDDの付いているシステムでも，プログラムサイズの小さい方がロード時間が短くなり，速度的な快適さが得られるはずだ．エディタやMAKEなどの子プロセスとして実行させるためには，実行時のメモリ消費量も最小に抑えたい．

こんな具合に，まだまだプログラムサイズを気にする時代は続いている．プログラムの設計，改良のときには，サイズの問題も頭の片隅で考慮して欲しい．

プログラムのサイズというと，最初に思い浮かぶのはプログラムのファイルサイズだが，本来，プログラムは3つのサイズを持っていることを忘れてはならない．スリーサイズ，といっても，

女性のB・W・H

ではない．ファイルサイズ，占有クラスタ数，メモリイメージのサイズの3つである．

ファイルサイズ

ファイルサイズは，一番簡単に確認できるプログラムのサイズだが，あまり意味のある数字ではない．もっとも，占有クラスタ数は，ファイルサイズと密接に関連しているので，それを知るための手がかりにはなってくれる．

占有クラスタ数

プログラムが，ディスク中で何クラスタ占有しているかで，このサイズが小さいほど，ディスクの使用効率が高くなる．また，ディスクキャッシュなどのソフトもクラスタ単位で処理を行っていることが多いので，占有クラスタ数を減らすことは，キャッシュの効率を上げることにもつながる．

ファイルサイズはバイト単位で小さくなるが，占有クラスタ数はクラスタサイズ分だけ小さくならないとまったく減らない．たとえば，1クラスタ8Kバイトのディスクでは，8193バイトのファイルと，16384バイトのファイルの占有クラスタ数は同じで，16384バイトのファイルを小さくするなら，8192バイト以上小さくしないと意味がない．「1000バイトも小さくなった」と喜ぶ前に，その結果占有クラスタ数が減ったかどうか，ちょっと確認してみて欲しい[*1]．

[*1] TOOLS-1に含まれるdiskstatコマンドでクラスタサイズを調べることができる．

面倒なことに，クラスタサイズはディスクによってまったく違う．MS-DOSの制限では，32バイト～64Kバイトまでの2のべき乗というだけだ．実際によく利用されているのは，512バイト(2DDやIBM系の2HD)，1Kバイト(8インチ2Dや5インチ/3.5インチ2HD)，2Kバイト(IBMのHDD)，4K/8K/16Kバイト(NECなどのHDD)だ．RAMディスクなどでは，128～256バイト/クラスタの場合もある．

これだけ多彩なパラメータが使用されていると，判断の目安がなくて困るのも事実だ．1K～2Kバイト単位で小さくできれば，多少は満足してもいいような気がする．

メモリイメージのサイズ

言葉のとおり，実行中に必要とされるメモリ量のことだ．これは，ファイルサイズとはあまり関係がない．簡単なプログラムだと，ファイルサイズが小さいほどメモリイメージも小さいことが多いのだが，ときには大量のワーク領域を必要とする場合もあるのでだまされないように注意して欲しい．

理想的なのは，利用可能なメモリ量に応じて，利用メモリ量を調整するプログラムだ．これなら，大量のメモリが使えるときには最高速で，メモリが少ないときにもそれなりに動作してくれる．

サイズと速度の微妙な関係

プログラムサイズと，実行速度には微妙な関連がある．まず，ファイルサイズは小さい方が望ましい．先に述べたとおり，ディスクからのロード時間などが短縮されるし，サイズの短縮を図った結果，無駄な処理をしているルーチンを発見できることもある．ただし，速度向上のために複雑な処理をしている場合には，サイズを犠牲に速度を稼いでいるのだから，例外だ．

メモリイメージについては，逆の傾向がある．速度向上を目指した結果，データ構造が簡潔になり，メモリ消費量が減ることもあるが，バッファリングやテーブルによる計算速度の向上などの手法でメモリを消費していることも多い．

結局，プログラムの使われ方を考えながら，速度とメモリ消費量のバランスを取るしかない．私の場合，フィルタ系のプログラムではファイルサイズに8Kバイト，または12Kバイトの制約を付け(処理内容の複雑さによって決める)，その範囲内で他のスペック(速度など)の向上に努力する，というアプローチを取ったことがある．

いつまでもあると思うな親と金

＜解説＞

マルチタスクOSでプログラムを書くときの鉄則．親とは親プロセスのことであり，金はハードウェア資源のことである．自分以外の他のプロセス（親プロセスも含めて）が，自分と同じように実行しているのか，あるいは必要なメモリはいつでも主記憶中に確保できるのかは定かではない．安全なプログラムを目指すなら，逐一確認するべきである．

キャウ・cat・キャウン

抽象的な話をいくら書いても，ピンとこない読者も多いだろうから，実際の開発を通して，プログラムのチューンアップを紙上体験して頂こう．もっとも，チューンアップの方向性は，あくまでも私自身のもので，読者によっては気に入らない部分も多いと思う．読者自身の価値観で，それぞれのアプローチを確立して欲しい．

サンプルとして取り上げるのはcatコマンド，ただしオプションは大幅に省略して−nのみを用意している．チューンアップを施すごとにベンチマークを行って，その結果を比べてみることにした．テストには，80386/25MHzのCPUを搭載したマシンで，102バイト/行×10000行(約1Mバイト)のファイルを10回，リダイレクトによってRAMディスク中でコピーした．−nオプションの有無により，2つの結果を載せてある．ベンチマークの結果と共に，ファイルサイズも併記した．

プログラム開発にはMS-C ver.6.00を使用し，すべてノーオプションでコンパイルを行った．他のバージョンでは，プログラムサイズや速度などで多少の違いが出るかもしれない．

なお，プログラムリストには，前のバージョンとの差分のみを掲載しているものがいくつかあるので，適宜補って欲しい．

普通のcatを書いてみる

最初に作るのは，何のてらいもない，普通のcatである(**リスト6-4**)．普通のプログラマが使い捨てのつもりで書いたら，こんな感じになるのではなかろうか．以降の改良の元となるバージョンだから，このバージョンを

「原始 cat」と呼ぶことにしよう.

リスト 6-4 「原始 cat」全リスト

```
#include <stdio.h>

#define TRUE    (1)
#define FALSE   (0)

#define LBUF    (1*1024)

int     progstat;
int     n_flag = FALSE;
int     count = 1;

void    filecopy(fp)
FILE    *fp;
{
        static char     buf[LBUF];

        while (fgets(buf, LBUF, fp) != 0)
                if (n_flag)
                        printf("%6d  %s", count++, buf);
                else
                        fputs(buf, stdout);
}

void    usage()
{
        fputs("Usage: cat [-n] [<filename>...]\n", stderr);
        exit(1);
        /*NOTREACHED*/
}

void    main(argc, argv)
int     argc;
char    **argv;
{
        FILE    *fp;

        progstat = 0;
        while (--argc > 0 && ***++argv == '-')
                switch (*++*argv) {
                case 'n':
                        n_flag = TRUE;
                        break;
                default:
                        usage();
                        /*NOTREACHED*/
                }
```

```
        if (argc == 0)
                filecopy(stdin);
        else
                for (; argc-- > 0; argv++) {
                        if ((fp = fopen(*argv, "r")) == NULL) {
                                perror(*argv);
                                progstat = 1;
                                continue;
                        }
                        filecopy(fp);
                        fclose(fp);
                }
        exit(progstat);
        /*NOTREACHED*/
}
```

プログラム名	サイズ（バイト）	実行時間（秒）	
		オプションなし	−n オプション
原始 cat	8891	69.59	121.99

表 6-2　「原始 cat」の性能

原始 cat はここが変

原始 cat を検討する前に，「何を改良するのか」をきちんと決めておこう．本章では，仕様の問題を最優先に，速度とサイズの最適化も図ることにする．

原始 cat の第一の問題点は，−n オプションが指定されたときに，「長い行を正しく処理できない」ということだ．プログラムでは，fgets()で 1 行取り込み，printf()または fputs()によって出力している．バッファより長い行が入力されると，行の途中に行番号が挿入され，それ以降の行番号も狂ってしまう．この問題が起きるのは，−n オプション指定時のみだ．オプションが指定されていない場合，fgets()で行が分断されても，その間にデータが追加されることがないので，fputs()の先では再び行が合体し，正し

い出力が得られる.

バッファサイズ(LBUF)を大きくすれば,この症状は出にくくなる.しかし,それでは根本的な解決になっていない(1Mバイトを超える行をどうやって処理するのか?).データの処理方法を,根本的に改める必要がある.

猫に旗を付けてみる

原始catの問題点(バグ)は,「fgets()で1行全体が取り込める」と勘違いしたことだ.元々,1行の長さについて,DOSは一切規定していないわけだし,fgets()だってそんなことは保証していない(「最大1行しか取り込めない」というのが正しい仕様)のだから,この考え方が間違っていたのは明らかである.

解決方法の1つは,fgets()で取り込んだデータの最後の文字がLFかどうかをチェックし,それによって次に取り込んだデータの前に行番号を付加するかどうかを決める方法だ.プログラムはリスト6-5のようになる.これを,「フラグ付きcat」と呼ぶことにしよう.

リスト6-5 「フラグ付きcat」差分リスト

```
void    filecopy(fp)
FILE    *fp;
{
        int     cont_line = FALSE;
        static char     buf[LBUF];

        while (fgets(buf, LBUF, fp) != NULL)
                if (n_flag) {
                        if (!cont_line)
                                printf("%6d  ", count++);
                        cont_line = (fputs(buf, stdout) != '\n');
                } else
                        fputs(buf, stdout);
}
```

プログラム名	サイズ（バイト）	実行時間（秒）	
		オプションなし	-n オプション
フラグ付き cat	8937	69.70	69.92

表 6-3 「フラグ付き cat」の性能

フラグ付き cat のいやらしいところは，データの処理にフラグを使っている，ということだ．フラグを利用したプログラムは，処理がどのように行われるかが読み取りにくくなる傾向がある．その結果，保守性に欠けるものになりがちだ．できることなら，プログラムの構造によって，フラグを使わないようにした方がいい．

逆に，フラグを使わないとすっきりした記述ができない場合もある．コマンドのオプションなどの管理がそれで，これをいちいち関数の引数として渡しているようでは，とても読めたプログラムにはならない．一般的にフラグは，プログラムの起動直後に設定し，それ以降は参照のみされるグローバル変数の形になるのが自然である．簡単なループ内部で頻繁に変更，参照しているような場合(フラグ付き cat のような)では，その周辺を作り直した方が，すっきりしたプログラムになる可能性もある．

フラグ付き cat は，プログラムの構造の面ではまだ検討の余地がありそうだ．速度，サイズの点も，まだ改善できるかもしれない．とはいえ，仕様どおりに動作するという点では一応使いものになるバージョンといえよう．

アプリケーション丸抱えへの道

とりあえず，仕様を満たすプログラムができあがった．ここで，プログラムのシェイプアップを目的に，低水準入出力による入出力を行ってみよう．fgets()や fputs()，printf()のような標準入出力ライブラリ(「高水準入出力」ともいう)の使用をやめて，read()や write()のような，より DOS の

ファンクションコールに近い，低水準入出力を使用することで，サイズや速度の面で，より完成度の高いプログラムにできるかもしれない．

printf()を使用しないので，行番号の文字列化には，sprintf()を使用する．この例でも明らかなように，「低水準」の名のとおり，read()/write()を使用する場合には，プログラム内部でかなりこまごまとした作業を行う必要がある．保守性が重視されるプログラムでは，多少性能を犠牲にしても，標準入出力ライブラリを使用した方が無難だ．

低水準入出力バージョンのcatは，リスト6-6のようになる．速度やサイズは表6-4のようになった．

リスト6-6　「低水準入出力 cat」全リスト

```
#include <memory.h>
#include <fcntl.h>
#include <stdio.h>

#define TRUE    (1)
#define FALSE   (0)

#define LBUF    (1*1024)

int     progstat;
int     n_flag = FALSE;
int     count = 1;

void    filecopy(fd)
int     fd;
{
        int     cnt;
        int     cont_line = FALSE;
        char    *p;
        char    *endp;
        static char     buf[LBUF];
        static char     lnumbuf[15];

        while ((cnt = read(fd, buf, sizeof(buf))) > 0) {
                if (!n_flag)
                        write(1, buf, cnt);
                else {
                        p = buf;
                        if (cont_line)
                                goto line_cont;
                        while (cnt > 0) {
                                sprintf(lnumbuf, "%6d  ", count++);
                                write(1, lnumbuf, 8);
```

```
line_cont:
                                endp = memchr(p, '\n', cnt);
                                if (endp == NULL) {
                                        cont_line = TRUE;
                                        endp = p + cnt;
                                } else {
                                        cont_line = FALSE;
                                        endp++;
                                }
                                write(1, p, endp - p);
                                cnt -= endp - p;
                                p = endp;
                        }
                }
        }
}

void    usage()
{
        fputs("Usage: cat [-n] [<filename>...]\n", stderr);
        exit(1);
        /*NOTREACHED*/
}

void    main(argc, argv)
int     argc;
char    **argv;
{
        int     fd;

        progstat = 0;
        while (--argc > 0 && ***++argv == '-')
                switch (*++*argv) {
                case 'n':
                        n_flag = TRUE;
                        break;
                default:
                        usage();
                        /*NOTREACHED*/
                }
        if (argc == 0)
                filecopy(stdin);
        else
                for (; argc-- > 0; argv++) {
                        if ((fd = open(*argv, O_RDONLY | O_TEXT)) < 0) {
                                perror(*argv);
                                progstat = 1;
                                continue;
                        }
                        filecopy(fd);
                        close(fd);
                }
        exit(progstat);
        /*NOTREACHED*/
}
```

プログラム名	サイズ(バイト)	実行時間(秒)	
		オプションなし	−nオプション
低水準入出力 cat	8003	44.27	96.18

表 6-4 「低水準入出力 cat」の性能

サイズの面では確かにコンパクトになっているが，速度的にはフラグ付き cat とあまり変わらない．特に，−n オプション指定時には，かなりの速度低下がみられる．高水準入出力のオーバーヘッドがなくなっているはずなのに，なぜ速くならないのだろうか？

低水準入出力のくせになぜ遅い?

上の例を見ても分かるように，低水準入出力を使えば必ず速くなる，とは限らない．図 6-1 を見ると分かるだろうが，高水準入出力では，適当なサイズのバッファを持っており，バッファが一杯になる(出力の場合)まで write()が呼び出されないのだ．入力の場合も同じことで，1 行が短い場合は，fgets()を何度も呼んでも，read()に制御が移らない．

つまり，低水準入出力を使うことで高速化しようと思ったら，プログラムで大きなバッファを用意し，一回当りのデータ転送量を増やし，入出力関数の呼び出し回数を抑えなければ意味がない．当然，複数行分のデータが一度に読み込まれるので，行の切り出しはプログラム内部で行う必要があるが，fgets()や printf()を使うオーバーヘッドはかなり削減できるし，データの転送単位も大きくなるので，かなりの高速化ができるはずだ．

ちなみに，標準入出力ライブラリのバッファサイズは，stdio.h の中で，定数 BUFSIZ として定義されている．MS-C の場合は 512 バイトがデフォルトである．

上のプログラムの場合，入力はまとめて読み込んでいるものの，書き出しは 1 行単位で write()を呼んでいる(−n オプション指定時)．行番号の出

力には，それとは別にwrite()を呼んでいるので，結局1行ごとに2回write()が呼ばれているわけだ．テストプログラムでは1行が102バイト，これに行番号8バイトが加わっても，1回のwrite()で55バイトしかデータを出力していないことになる．

標準入出力を使用した場合(フラグ付きcatなど)，バッファが一杯になるまで，つまり512バイトのデータがたまるまでwrite()は呼ばれない．これと比べると，上のプログラムは，write()の呼び出し回数が9.3倍多いことになる．

一方，入力(read()の呼び出し回数)はどれだけ改善されたのだろうか．標準入出力ライブラリを使用すると，512バイトごとに一回read()が呼ばれる．それに対して上のプログラムでは，バッファサイズを1024バイトに設定しているので，read()の呼び出し回数は1/2に減ったことになる．単純に考えると，データ1024バイトごとのread()/write()の使用回数は，標準入出力ライブラリ使用時には4回(2+2)，リスト6-6では19.6回(1+18.6)になる．これでは，速くならないのは当然といえよう．

このように考察してみると，プログラムの速度低下の主因は，出力のバッファリングを行っていないことで，入力に関してはある程度の性能改善がなされている．したがって，出力にのみ標準入出力ライブラリを利用すれば，プログラムは高速になるはずだ．

リスト6-7と**表6-5**に，出力に標準入出力ライブラリfwrite()を使用した，「出力バッファリングcat」のソースとその性能を示す．低水準入出力catと比べると，−nオプション指定時の処理速度が，3割ほど上がっていることがわかる．

リスト 6-7　「出力バッファリング cat」差分リスト

```
void    filecopy(fd)
int     fd;
{
        int     cnt;
        int     cont_line = FALSE;
        char    *p;
        char    *endp;
        static char     buf[LBUF];
        static char     lnumbuf[15];

        while ((cnt = read(fd, buf, sizeof(buf))) > 0) {
                if (!n_flag)
                        write(1, buf, cnt);
                else {
                        p = buf;
                        if (cont_line)
                                goto line_cont;
                        while (cnt > 0) {
                                sprintf(lnumbuf, "%6d  ", count++);
                                fwrite(lnumbuf, 8, 1, stdout);
line_cont:
                                endp = memchr(p, '\n', cnt);
                                if (endp == NULL) {
                                        cont_line = TRUE;
                                        endp = p + cnt;
                                } else {
                                        cont_line = FALSE;
                                        endp++;
                                }
                                fwrite(p, endp - p, 1, stdout);
                                cnt -= endp - p;
                                p = endp;
                        }
                }
        }
}
```

プログラム名	サイズ（バイト）	実行時間（秒）	
		オプションなし	-n オプション
出力バッファリング cat	8019	44.27	72.55

表 6-5　「出力バッファリング cat」の性能

こうすればあなたも痩せられる

プログラムをさまざまな視点から評価，改良する訓練は，決して無駄にはならないだろうし，改良の基準を変えて行くことで，プログラムの構造がすっきりし，もとの改良(この場合は速度向上)が楽になることもままあることだ．そこで，速度向上は一休みして，プログラムサイズの短縮に目を向けることにしよう．

もちろん，四方八方手を広げた挙句に，プログラムはぐちゃぐちゃ，性能は一向に向上せず，という可能性もある．ソースの状態をチェックしながら，何を目標に次の作業を行うか，常に考えておきたいものだ．

ここでは，プログラムのファイルサイズ(と占有クラスタ数)に注目して小型化を図ることにする．出力バッファリング cat のサイズは 8019 バイトで，8K バイト(＝8192 バイト)をわずかに切った程度である．これは，ワイルドカードの展開ルーチン(setargv.obj)をリンクしていない状態だ．実際のリリースバージョンでは必ず setargv()をリンクするだろうから，もう少し小さくしておきたい．具体的なサイズを挙げると，setargv()抜きで 6K バイト程度まで小さくできれば，setargv.obj や今後の改良を付加しても，最終的に 8K バイト以下のファイルサイズを実現できるだろう．

ファイルサイズを小さくするにはいくつか方法があるが，ここでは「リンクされるライブラリを最小に抑える」という方針でいく．現状ではソースプログラムも小さく，データ(初期化データ)も少ないため，プログラム本体の小型化を目指しても，たかがしれているからだ．

プログラムに，どのようなライブラリ関数がリンクされているかを調べる際には，リンカのマップファイルが役に立つ．図 6-6 は，出力バッファリング cat のマップファイルの一部だ．シンボルがアドレス順に出力されている欄に注目すると，どのライブラリが大きいか，すぐ分かる．

```
0000:09B2       __stbuf
0000:0A23       __ftbuf
0000:0A62       _fflush
0000:0AD4       _flushall
0000:0B36       __output
0000:100C       _close
0000:102C       _lseek
0000:10A6       _sopen
0000:10C0       _open
0000:1251       __cXENIXtoDOSmode
0000:1262       _read
0000:134C       _write
0000:148A       _stackavail
```

図6-6 「出力バッファリングcat」のマップファイル(一部)

このプログラムの場合は，__outputという関数が1Kバイト以上を占めており，けっこう大きい．実はこの関数，printf()やfprintf()/sprintf()の下請け関数，というよりはその本体だ(この辺りの情報は，ライブラリのソースを入手し通読するか，ライブラリを逆アセンブルするとよく分かる)．

現在sprintf()は，行番号の文字列化のために使用しているだけで，この部分だけを書き直せばよい．Cのライブラリには，int型をASCII文字列に変換するitoa()なる関数が用意されているので，これを使ってsprintf()の使用をやめてみよう．

ついでにもう1つ，usage関数中で，メッセージの出力にfputs()を使用しているが，これもやめることにしよう．プログラム本体(filecopy関数)で使用している標準入出力はfwrite()のみで，fputs()はここでしか使用していない．したがって，これを書き替えるとさらに小さくなるはずだ．

サイズの小型化を目指した「小型化cat」のソースを**リスト6-8**に，速度等を**表6-6**に示す．サイズについては，そこそこ小さくなったとはいえ，目標の6Kバイトには届かなかったようだ．面白いことに，-nオプション

指定時の速度が，多少速くなっている．これは，sprintf()のオーバーヘッドがなくなったためであろう．やはり，プログラム開発では視野を広く持つことが肝要だという，

これは天のお告げ

ではないだろうか．

リスト6-8　「小型化cat」差分リスト

```
#define NFIGURE         (6)
#define NMAXLSPACE      (NFIGURE - 1)
#define NSPACE          (2)
#define NFIELD          (NFIGURE + NSPACE)

#define outnum()        {\
        int     len;\
        memset(lnumbuf, ' ', NMAXLSPACE);\
        itoa(count++, lnumbuf + NMAXLSPACE, 10);\
        len = strlen(lnumbuf + NMAXLSPACE);\
        lnumbuf[NMAXLSPACE + len + 0] = ' ';\
        lnumbuf[NMAXLSPACE + len + 1] = ' ';\
        fwrite(lnumbuf+NMAXLSPACE+len-NFIGURE, NFIELD, 1, stdout);\
}

void    filecopy(fd)
int     fd;
{
        int     cnt;
        int     cont_line = FALSE;
        char    *p;
        char    *endp;
        static char     buf[LBUF];
        static char     lnumbuf[NFIELD + NMAXLSPACE];

        while ((cnt = read(fd, buf, sizeof(buf))) > 0) {
                if (!n_flag)
                        write(1, buf, cnt);
                else {
                        p = buf;
                        if (cont_line)
                                goto line_cont;
                        while (cnt > 0) {
                                outnum();
```

```
line_cont:
                                endp = memchr(p, '\n', cnt);
                                if (endp == NULL) {
                                        cont_line = TRUE;
                                        endp = p + cnt;
                                } else {
                                        cont_line = FALSE;
                                        endp++;
                                }
                                fwrite(p, endp - p, 1, stdout);
                                cnt -= endp - p;
                                p = endp;
                        }
                }
        }
}

void    usage()
{
        static char E_USAGE[] = "Usage: cat [-n] [<filename>...]\n";

        write(2, E_USAGE, sizeof(E_USAGE) - 1);
        exit(1);
        /*NOTREACHED*/
}
```

プログラム名	サイズ(バイト)	実行時間(秒)	
		オプションなし	-n オプション
小型化 cat	6517	44.27	63.60

表 6-6　「小型化 cat」の性能

テキストモードとバイナリモード

MS-C のライブラリでは，入出力に「テキストモード」と「バイナリモード」の 2 つがあることは御存じだろう．実はこれ，入出力の速度にかなり効いてくる．

バイナリモードのとき，ライブラリ(read() と write())はデータにはまったく手を加えない．DOS から渡されたデータをそのままアプリケーションに渡し，アプリケーションから渡されたデータをそのまま DOS に渡す，

と，ただの卸問屋のようなことをしている．

それに対してテキストモードでは，ライブラリはデータを加工して，それぞれの環境間の橋渡しをする．つまり，read()はLFの直前のCRと^Z以降のデータを削除し，逆にwrite()はLFの直前にCRを挿入する．先ほどの卸問屋に対比させると，VAR(付加価値業者)のようなものといえる．

卸問屋より，VARの方がマージンが高いのと同様に，テキストモードの方がバイナリモードよりもコストが高い．つまり，遅いのである．処理の内容を考えると当然のことといえるだろう．

バイナリモードにするとどれくらい速くなるかは，簡単に確かめられる．main()中のopen()の引数，O_TEXTをO_BINARYに変え，setmode()によってファイルハンドル0と1をバイナリモードに設定するだけだ．**リスト6-9**と**表6-7**に，小型化catのバイナリバージョン(「バイナリcat」)のソースと性能を記す．どうです，速くなったでしょう？

リスト6-9　「バイナリcat」差分リスト

```
void    main(argc, argv)
int     argc;
char    **argv;
{
        int     fd;

        setmode(0, O_BINARY);
        setmode(1, O_BINARY);
        progstat = 0;
        while (--argc > 0 && ***++argv == '-')
                switch (*+++argv) {
                case 'n':
                        n_flag = TRUE;
                        break;
                default:
                        usage();
                        /*NOTREACHED*/
                }
        if (argc == 0)
                filecopy(stdin);
        else
```

```
        for (; argc-- > 0; argv++) {
                if ((fd = open(*argv, O_RDONLY | O_BINARY)) < 0) {
                        perror(*argv);
                        progstat = 1;
                        continue;
                }
                filecopy(fd);
                close(fd);
        }
    exit(progstat);
    /*NOTREACHED*/
}
```

プログラム名	サイズ(バイト)	実行時間(秒)	
		オプションなし	-n オプション
バイナリ cat	6645	15.55	30.15

表 6-7　「バイナリ cat」の性能

オプションなしのときで，小型化 cat 等と比べて，25 %ほど高速になっている．言い換えると，read()や write()の変換だけで，処理時間の 25 %を消費していたわけだ．

ただし，この変更を加えると，プログラムの挙動が，微妙に違ってくる．たとえば，EOF(^Z)が現れても一般の文字としてアプリケーションに渡されてしまうし，CR/LF の扱いも微妙に違ってくる．まあ，MS-DOS 環境下で作成した普通のテキストファイルを扱う分には違いもあまり目立たないし，ユーザーによっては「透過性が向上した」と喜ぶかもしれないが，性能向上が一段落したら，きちんと仕様を詰めておくべきだろう(本書ではやらないが)．

もっと^2 速くしたい!

出力バッファリングバージョン(表6-5)以降のプログラムでは，標準入出力ライブラリによって出力をバッファリングしていた(−nオプション指定時のみ)．これはこれで高速化に貢献しているのだが，より高速なプログラムにしようと思ったら，出力のバッファリングも自前で，つまりプログラム内部で行う方がよい．標準入出力ライブラリを使用すると，それらの関数がリンクされる分プログラムサイズも大きくなるし，入出力の経路が複雑になることで，速度低下も免れないからだ．

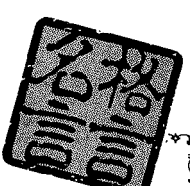
バッファリングの概念図が，図6-7のようになることは，言うまでもないだろう．標準入出力ライブラリなどが，この方式で設計されているものの一例だ．

図6-7　一般的なバッファリングの概念図

この方法の問題点は，プログラムのワーク領域とバッファ領域を別々に用意する必要がある，ということだ．プログラムの実行イメージの大きさ(必要メモリ量)は少ない方がいいに決まっている．逆に，一定の領域をできる限り有効に利用するためには，ワーク領域とバッファ領域を融合する方が望ましい．せっかくプログラム内部でバッファリングまで抱え込むのだから，そこまでやってもいいのではないだろうか．

次に作成する「自前バッファリングcat」では，catがフィルタ系のプログラムであることを利用して，単一領域をバッファとワーク領域両方に使っている．図6-8は，catでバッファ領域とワーク領域を合体させた場合の管理方法の概念図だ．catは，データの処理中に行番号などを付加する場合がある．つまり入力データより出力データの方が多い可能性がある．そこで，データを読み込むときに，バッファの先頭ではなく，途中から読み込むようにする．データを加工しながら，出力データをバッファの先頭に詰めていくと，入力データが処理されていくので，兎と亀の追いかけっこのように，2種類のデータの境界(入力データの先頭と出力データの末尾)が移動していく．

行番号を付加している場合，途中で出力データの末尾が入力データの先頭に追いつく可能性がある．そこで，2つのポインタを常時チェックして，追いつきそうになったら，出力データをフラッシュ(実際に出力すること)する．これを繰り返せば，最後にはバッファ中の全データが処理できるわけだ．



諸行無常

＜解説＞

プログラム中のどの行も，常識外れのコーディングがしてある様子．

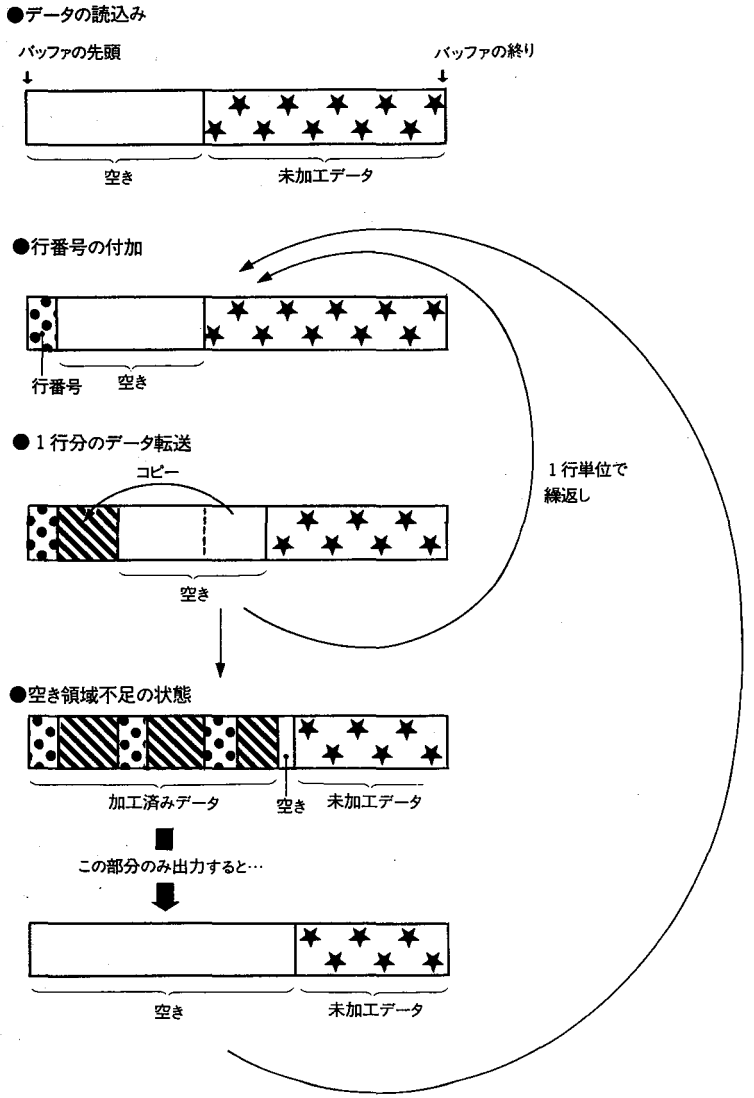


図 6-8　自前バッファリング cat のバッファ管理

もちろん，オプションが指定されていない場合には，このような処理は不要であり，バッファ目一杯にデータを読み込み，それをそのまま write() で書き出している．リスト 6-10 と表 6-8 は，自前バッファリング cat のプログラムと性能だ．

リスト 6-10　「自前バッファリング cat」差分リスト

```
#define R_OFF   (LBUF / 5 * 2)

int     progstat;
int     n_flag = FALSE;
int     count = 1;

#define NFIGURE         (6)
#define NMAXLSPACE      (NFIGURE - 1)
#define NSPACE          (2)
#define NFIELD          (NFIGURE + NSPACE)

#define outnum()        {\
        int     len;\
        itoa(count++, lnumbuf + NMAXLSPACE, 10);\
        len = strlen(lnumbuf + NMAXLSPACE);\
        lnumbuf[NMAXLSPACE + len + 0] = ' ';\
        lnumbuf[NMAXLSPACE + len + 1] = ' ';\
        memcpy(wp, lnumbuf+NMAXLSPACE+len-NFIGURE, NFIELD);\
        wp += NFIELD;\
}

#define flush_writebuf() (write(1, buf, wp - buf), wp = buf)

void    filecopy(fd)
int     fd;
{
        int     offset;
        int     cnt;
        int     cont_line = FALSE;
        char    *p;
        char    *endp;
        char    *wp;
        static char     buf[LBUF];
        static char     lnumbuf[15];

        memset(lnumbuf, ' ', sizeof(lnumbuf));
        offset = n_flag ? R_OFF : 0;
```

```
        while ((cnt = read(fd, (p = buf+ offset), sizeof(buf) - offset)) > 0) {
                if (!n_flag)
                        write(1, p, cnt);
                else {
                        wp = buf;
                        if (cont_line)
                                goto line_cont;
                        while (cnt > 0) {
                                outnum();
line_cont:
                                endp = memchr(p, '\n', cnt);
                                if (endp == NULL) {
                                        cont_line = TRUE;
                                        endp = p + cnt;
                                } else {
                                        cont_line = FALSE;
                                        endp++;
                                }
                                memcpy(wp, p, endp - p);
                                wp += endp - p;
                                cnt -= endp - p;
                                p = endp;
flush_buf:
                                if (p - wp < NFIELD)
                                        flush_writebuf();
                        }
                        flush_writebuf();
                }
        }
}
```

プログラム名	サイズ（バイト）	実行時間（秒）	
		オプションなし	−n オプション
自前バッファリング cat	5857	15.54	33.89

表 6-8　「自前バッファリング cat」の性能

バイナリ cat と比べると，−n オプション指定時に 1 割ほど遅くなっているが，オプションがない場合の速度は変わらない．ただし，サイズの点でもかなりの改善がみられる．これは，標準入出力ライブラリをまったく使わない状態にしたことで，リンクされるライブラリが相当減ったことを意味している．

さて，−n オプション指定時に，バイナリ cat の方が速いのはなぜか，というと，自前バッファリングバージョンでは，read()時にバッファの一部(プログラムでは 3/5)にしかデータを読み込まず，したがって read()の呼び出し回数が増えてしまうのが理由だ．これを解決するのは簡単で，バッファサイズそのものを大きくすれば，1回の読み書きで大量のデータが転送されるようになり，転送回数が相当減る．バイナリcatでは出力に標準入出力ライブラリを使用しているので，プログラム内部のバッファをいくら大きくしても，read()の回数が減るだけで，write()の回数は大して減少しない．

表 6-9 は，バッファサイズを 16K バイトにしたとき(LBUF の定義を変えたとき)の，バイナリ cat と自前バッファリング cat の性能だ．−n オプション指定時の速度が逆転し，自前バッファリング cat のメリットがはっきりと現れている．フラグ付き cat の値と比べてみて欲しい．プログラム改良前に対して，どの程度速度，サイズが向上したか，そのためにプログラムがどの程度複雑になったかを検討してみて欲しい．改良の結果は，作業の手数に見合うものだっただろうか？この速度，サイズは，ユーザーの要求を満たすのに十分だろうか？

プログラム名	サイズ(バイト)	実行時間(秒)	
		オプションなし	−n オプション
バイナリ cat (16K バイト)	6645	8.02	27.35
自前バッファリング cat(16K バイト)	5587	8.02	19.01

表 6-9　「バッファサイズ 16K バイト」の性能

私の価値観からすると，バッファサイズ 16K バイトの自前バッファリングバージョンで十分だと判断し，速度，サイズの追及をやめることにした．この辺りはプログラマによって合格ラインが違うだろうから，読者の満足するまで追及してみてもらたい．**リスト 6-11** に，自前バッファリング cat の完全なソースを掲載しておく．

リスト6-11 「自前バッファリングcat(16Kバイトバージョン)」全リスト

```
#include <memory.h>
#include <fcntl.h>
#include <stdio.h>

#define TRUE    (1)
#define FALSE   (0)

#define LBUF    (16*1024)
#define R_OFF   (LBUF / 5 * 2)

int     progstat;
int     n_flag = FALSE;
int     count = 1;

#define NFIGURE         (6)
#define NMAXLSPACE      (NFIGURE - 1)
#define NSPACE          (2)
#define NFIELD          (NFIGURE + NSPACE)

#define outnum()        {\
        int     len;\
        itoa(count++, lnumbuf + NMAXLSPACE, 10);\
        len = strlen(lnumbuf + NMAXLSPACE);\
        lnumbuf[NMAXLSPACE + len + 0] = ' ';\
        lnumbuf[NMAXLSPACE + len + 1] = ' ';\
        memcpy(wp, lnumbuf+NMAXLSPACE+len-NFIGURE, NFIELD);\
        wp += NFIELD;\
}

#define flush_writebuf() (write(1, buf, wp - buf), wp = buf)

void    filecopy(fd)
int     fd;
{
        int     offset;
        int     cnt;
        int     cont_line = FALSE;
        char    *p;
        char    *endp;
        char    *wp;
        static char     buf[LBUF];
        static char     lnumbuf[15];
```

```
        memset(lnumbuf, ' ', sizeof(lnumbuf));
        offset = n_flag ? R_OFF : 0;
        while ((cnt = read(fd, (p = buf+ offset),
                                sizeof(buf) - offset)) > 0) {
                if (!n_flag)
                        write(1, p, cnt);
                else {
                        wp = buf;
                        if (cont_line)
                                goto line_cont;
                        while (cnt > 0) {
                                outnum();
line_cont:
                                endp = memchr(p, '\n', cnt);
                                if (endp == NULL) {
                                        cont_line = TRUE;
                                        endp = p + cnt;
                                } else {
                                        cont_line = FALSE;
                                        endp++;
                                }
                                memcpy(wp, p, endp - p);
                                wp += endp - p;
                                cnt -= endp - p;
                                p = endp;
flush_buf:
                                if (p - wp < NFIELD)
                                        flush_writebuf();
                        }
                        flush_writebuf();
                }
        }
}

void    usage()
{
        static char E_USAGE[] = "Usage: cat [-n] [<filename>...]\n";

        write(2, E_USAGE, sizeof(E_USAGE) - 1);
        exit(1);
        /*NOTREACHED*/
}

void    main(argc, argv)
int     argc;
char    **argv;
{
        int     fd;

        setmode(0, O_BINARY);
        setmode(1, O_BINARY);
        progstat = 0;
        while (--argc > 0 && **++argv == '-')
```

```
                switch (*++*argv) {
                case 'n':
                        n_flag = TRUE;
                        break;
                default:
                        usage();
                        /*NOTREACHED*/
                }
        if (argc == 0)
                filecopy(stdin);
        else
                for (; argc-- > 0; argv++) {
                        if ((fd = open(*argv, O_RDONLY | O_BINARY)) < 0) {
                                perror(*argv);
                                progstat = 1;
                                continue;
                        }
                        filecopy(fd);
                        close(fd);
                }
        exit(progstat);
        /*NOTREACHED*/
}
```

あとは野となれ山となれ

＜解説＞

プログラマが退職時に必ず心の中でつぶやく言葉.

仕様を煮詰める

リスト6-11のcatは，高速性の点ではかなり優秀な部類に入るものだと思う．これ以上の高速化を求めると，部分的にアセンブラによって記述したり，コードの一文一文を検討して細かなチューンを行う必要があるだろう．バッファサイズを大きく(64Kバイト近く)にしたり，バッファサイズと読み込みデータサイズの比(現在は3/5)を調整するのも方法の1つだ．

速度の面では合格点を付けるとしても，このプログラムは，仕様の面からいうと完成度は非常に低く，リリースするのがためらわれるのも事実だ．バイナリcat(リスト6-9)の項で説明したとおり，EOF(^Z, '¥x1a')やCR(^M, '¥x0d')の扱いには統一性がないし，エラーチェック(read()やwrite()，open()など)やメッセージの出力なども不十分だ．-n以外のオプションを追加しようとした場合の拡張性にも問題があるだろう．本節では，紙上プログラミング体験のまとめとして，仕様にまつわる話題を取り上げることにしよう．

バイナリモードは採用すべきか

バイナリモードでテキストを扱う場合の問題点はすでに説明したが，この辺りをきちんとサポートしようとすると，けっこう複雑な処理が必要になり，速度，保守性の両方が低下し，そのメリット(高速性)を十分に引き出せない可能性がある．私も一時期，自作ツールを片端からバイナリモード方式に改造したことがあるが，1～2年ほど放って置いたところ，何をやっているのか自分でも分からなくなり，慌ててテキストモードを使うように戻した経験がある．現在では，プログラムの仕様上，入力データを出力

する必要のないコマンド(wcなど)を除けば，テキストモードによる入出力を使用するようになっている．

読者も，どちらの方式を取るかを決めるときには，自分の力量やプログラミングに割ける時間，そのプログラムの寿命なども慎重に検討することをお勧めする．

スタックオーバーフローとの闘い

スタックオーバーフローなんていうものは，再帰的な関数呼び出しを行なっているプログラムではよく起こる事故であるが，人によってはバグの一種に分類することもあるし，ユーザー(ソースファイルを持っていない人)は場当たり的な対処しかできないので注意しておきたい事柄の1つといえる．

MS-Cというと，昔は「すぐにスタックオーバーフローを起こす」処理系として有名(?)だったが，最近はそんな噂も聞かなくなった．実は，MS-C ver.3.00.17では，環境変数をプログラムのスタック領域にコピーしている．スタック領域はデフォルトで2Kバイトしかなかったため，環境変数をたくさん設定しているシステムでは，簡単にスタックオーバーフローが発生したのである．TOOLSのプログラマは，この問題に真っ向から取り組み，単純かつ効果的な対処方法を発見したので，報告させて頂こう．MS-Cも，ver.4から環境変数のコピー先をヒープ領域に変更したため，環境変数が原因のスタックオーバーフローは起きなくなったが，この対処方法にはオーバーフロー防止以外の効用もあるので，安心して読んで欲しい．

スタートアップが環境変数をスタック領域(またはヒープ領域)にコピーするのには，2つの理由がある．1つは，環境を変更可能にするため，つまりputenv()を実現するためである．つまり，putenv()を使用しないプログラムなら，コピーは必要ない．

もう1つの理由は，スモールモデルやミディアムモデルで，環境変数をnearポインタで指せるようにするためだ．DOSでは，プロセスの環境変数

が独立したセグメントに格納されているため，そのままでアクセスしようとするとfarポインタが必要になる．逆にいうと，farポインタを明示的に使用するようにすれば，そのまま環境変数を参照できる．

環境変数のコピーは，_setenvp()という関数で行っている．そこで，ソースファイル中に次のような定義を含めてやると，環境変数のコピーが行われなくなる．

```
#ifdef MSC
_setenvp()
{
}
#endif /* MSC */
```

ただし，識別子"MSC"はMS-Cの定義済み識別子ではないので，コンパイル時にはきちんと指定して欲しい．また，リンカによっては/noeオプションを指定しないとうまく行かない場合がある．

この5行をおまじないのつもりでソースファイルに含めておけば，環境変数のコピーがなされないため，スタックやヒープの領域が無駄遣いされることはなくなる．さらに，_setenvp()も小さくなるため，EXEファイルのサイズも減少する．MS-C ver.6.00を例に取ると，これによって600バイト程度ファイルサイズが小さくなった．また，スタートアップの処理が減る分，プログラムの実行速度も向上する．

いいことづくめのようだが，普通のgetenv()やputenv()，グローバル変数environなどが使えなくなることには注意して欲しい．_setenvp()がコピーした環境変数は，exec()やspawn()，system()などの子プロセス起動関数からも参照されているので，これらの関数も正常には使えなくなる(子プロセスに，環境変数が引き継がれなくなる)．

上のコードを取り込んだプログラムで，環境変数を参照したければどうすればいいのか．MS-Cでは，メモリモデルに関係なくfarポインタを利用

できるので，それを利用して，「far ポインタを返す getenv()」を用意すればよい．リスト 6-12 は，環境変数を検索して far ポインタで返す関数，_fgetenv()の例である．アセンブラで書けばもう少し小さくなるので，愛用する場合にはチャレンジしてみてもらいたい．

リスト 6-12　_fgetenv()関数

```
#ifndef lint
static char sccsid[] = "@(#)_fgetenv.c  1.1 3/10/88";
#endif

/*
 * _fgetenv.c - get environment variable
 */

#include <stdio.h>
#include <dos.h>

#define OFFSET_ENV      (0x2c)

char far *_fgetenv(name)
char *name;
{
        extern unsigned int _psp;
        char far *p;
        register char *np;

        FP_SEG(p) = _psp;
        FP_OFF(p) = OFFSET_ENV;
        FP_SEG(p) = *((unsigned int far *)p);
        FP_OFF(p) = 0;

        while (*p != '\0') {
                for (np = name; *np != '\0'; np++, p++)
                        if (*np != *p)
                                break;
                if (*np == '\0' && *p == '=')
                        return p + 1;
                while (*p != '\0')
                        p++;
                p++;
        }
        return NULL;
}
```

```
#ifdef TEST
int     progstat = 0;

void    main(argc, argv)
int     argc;
char    **argv;
{
        char    far *p;

        while (--argc > 0) {
                argv++;
                if ((p = _fgetenv(*argv)) == NULL) {
                        printf("%s is not defined.\n", *argv);
                        progstat = 1;
                } else
                        printf("%s = %Fs\n", *argv, p);
        }
        exit(progstat);
        /*NOTREACHED*/
}
#endif

/* ---- end of _fgetenv.c ---- */
```

エラーメッセージも大事だよ

エラーメッセージの仕様，つまり，どのようなメッセージをどのような順番で表示するのかも，立派なユーザーインターフェイスだ．最も簡単なのは，おそらくエラーメッセージを表示せず，黙って処理を中断(exit)するものであろう．これより多少親切なメッセージとして，〝?〟や〝error〟といった文字列を表示して処理を中断するものもある．

これらの方法の困った点は，プログラムのソースを持っているユーザーにしか，どんなエラーが起きたのかを知るすべがないということだ．一般ユーザーのことを考えると，もう少し丁寧なメッセージを表示したいものだ．

Cのライブラリには，perror()というエラーメッセージの表示関数が用意されており，簡単に，定形のエラーメッセージを出力できるのは，読者も御存じだろう．perror()の使い方には，3つの流儀がある．

1つ目は，〝perror(*command_name*)〟の形で，どのコマンドでエラーが

起きたかを知らせるというものだ．フィルタ系のプログラムなどでは，パイプによって1コマンドラインで複数のコマンドを起動できる．そのうち，どのコマンドでエラーが発生したのかを知らせようという考え方が，この方式の基本だ．

2つ目は，〝perror(*filename*)〟の形で，どのファイルを処理中にエラーが発生したかを知らせようとするものだ．パイプを使っている場合はともかく，たいていの場合にはなかなか優れたやり方といえる．

3つ目の流儀では，コマンド名と処理中のファイル名両方を表示する．多少表示は長くなるが，ユーザーにとっては，一番丁寧なものといえるだろう．この場合，一般的なメッセージフォーマットは次のようになる．

command_name：*filename*：*message*

さて，エラーメッセージの先頭に表示するコマンド名は，文字列定数として，プログラム埋込みになっていることが多い．たとえば，前節で取り上げたcatでは，usageの表示文字列の先頭に，〝cat：〟という文字列が埋まっていた．

この方法は，プログラムを簡潔に保つのには都合がよいのだが，ユーザーにとっては不便なこともある．コマンド名(EXEやCOMのファイル名)を変更すると，コマンド名とメッセージの内容が矛盾するからだ．

したがって，プログラムではコマンドファイル名(mainの引数として，argv[0]で与えられる)を解析し，エラーメッセージではそこから得られたコマンド名を表示すべきである．ただし，argv[0]でファイル名が渡されるのは，DOSのバージョンが3.0以上の場合のみなので，DOS 2.*x*の場合は，諦めて埋込みのコマンド名を使わざるを得ない．

TOOLS-3などのコマンドの中には，こういった処理を，setprog()という関数で行っているものもある．**リスト6-13**にsetprog()のソース(サンプルプログラム込み)を示す．

リスト6-13　setprog()のソース

```
#ifndef lint
static char sccsid[] = "@(#)setprog.c   1.1 4/26/88";
#endif

#include <stdio.h>
#include <ctype.h>
#include <assert.h>
#include <dos.h>

char progname[64];
extern  char    *jstrrchr();
extern  char    *jstrlwr();

#ifdef KANJI
static char     *jstrlwr(str)
char    *str;
{
        register char   *s;

        for (s = str; *s != '\0'; )
                if (iskanji(*s) && iskanji2(s[1]))
                        s += 2;
                else {
                        *s = tolower(*s);
                        s++;
                }
        return str;
}
#else
#define jstrlwr         strlwr
#define jstrrchr        strrchr
#endif

void    setprog(arg0, defname)
char    *arg0;
char    *defname;
{
        extern unsigned char _osmajor;
        register char   *q;
        register char   *p;

        if (_osmajor <= 2)
                strcpy(progname, defname);
        else {
                if (isalpha(*arg0) && arg0[1] == ':')
                        arg0 += 2;
                if ((p = jstrrchr(arg0, '/')) != NULL)
                        p++;
                else
                        p = arg0;
```

```
                if ((q = jstrrchr(arg0, '\\')) != NULL)
                        q++;
                else
                        q = arg0;
                if (q > p)
                        p = q;
                strcpy(progname, p);
                for (p = progname; *p != '\0' && *p != '.'; ) {
#ifdef KANJI
                        if (iskanji(*p) && iskanji2(p[1]))
                                p++;
#endif
                        p++;
                }
                *p = '\0';
                jstrlwr(progname);
        }
}

#ifdef TEST
void    main(argc, argv)
int     argc;
char    **argv;
{
        setprog(argv[0], "def_name");
        printf("argv[0] = %s\nprognam = %s\n", argv[0], progname);
}
#endif
```

こんな機能をサポートしても，一般ユーザーは気にもせず，気づいても感心してはくれないだろう(「その分プログラムを小さくしろ」を怒るかもしれない)が，プログラマが一種のサービス業(ユーザーに対する)である以上，こういった気遣いも大切だ(大切かもしれない)．

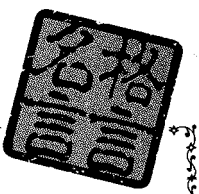
また，エラーメッセージは，perror()のメッセージ(グローバル変数，char * sys_errlist[]中に格納されている)をそのまま使用するのが基本になるが，いくつか考慮すべき問題もある．まず，sys_errlist 中のメッセージが妥当かどうか，ということ．そして，プログラム固有のエラーメッセージをどうするか，ということだ．

MS-C のライブラリは，UNIX のものをかなり忠実に(DOS で可能な範囲で，だが)模倣している．エラーメッセージも同様で，メッセージの種類は，基本的に UNIX のもののサブセットになっている．

しかし，DOS は UNIX のサブセット環境ではない．たとえば，〝Drive Not Ready〟エラーは，随時交換可能なディスクがサポートされている DOS 環境特有のもので，UNIX には相当するエラーの概念がない．

このような，DOS 特有のエラーまで，UNIX のエラーメッセージに無理やりマップしているため，perror()を信じきっていると，時々とんちんかんなメッセージが表示されることがある．プログラムのユーザー層によっては，自前でエラーメッセージを用意した方がよいだろう．

プログラム固有のエラーとは，行の長さ制限や，1 行当りのフィールド数などの制限を超えた場合のことだ．これらのエラーに対応するメッセージは，perror()には用意されていないので，それらのメッセージはプログラム内部で抱え込む必要がある．



早起きな左門豊作

＜解説＞

古くは「早起きは三文の得」．農家の親戚に預けられた左門豊作は，早起きをして農作業に精を出したからこそ強打者に成長した．プログラマも夜型の生活から脱却し，早起きしてすっきりした頭でプログラミングすれば，三文プログラマから天才プログラマへと一気に飛躍できる，というありがたい教え．

ウィザードへの道はまだまだ長いぞ

さて，この章ではcatの開発を例にとって，いかにプログラムの完成度を高めるかについて述べてきた．読者の方々も，同じ目的のプログラムでも，作り方によってどれほど差がつくか，おわかりいただけたのではないだろうか．

多人数で1つのプログラムを完成させるような場合には，プログラムの完成度にはプロジェクトリーダーの才覚が大きくものをいうことになるのだろうが，個人のプログラミングの場合には，多少の才能の違いもあるにせよ，つまるところは，

努力，根性，粘り強さ，向上心

に集約される(言い方を換えれば，単に「個人の性格」だったりして)．とか言うと，なんだか中学校時代のホームルーム(「学活」とも言う)や道徳の時間みたいだが，結局，一番の違いは，

どれだけプログラミングが好きか

によるのだろう．その情熱を正しく導いてくれるのが，本書のような(?)異端の書籍だったり，先輩プログラマの「ダメ」「タコ」「大名」といった，心のこもった激励の言葉なのである．

この本の内容を完全に身に付けたとしても，ウィザードへの道のほんのとば口に立ったに過ぎない．やがて『続・実録！天才プログラマ』にでも紹介されるようになるまで，お互い不断の努力を続けたいものである．

■著者略歴

福崎俊博(ふくざき　としひろ)

1958年1月　東京生まれ
葛飾柴又育ち，水瓶座，血液型はいまだに不明．京成自動車学校第一種普通免許教習課程中退．大学時代，オーディオ誌に載っていたPET 2001の広告を見てコンピュータに興味を持つが，なぜかApple IIを購入．趣味が高じて某コンピュータ会社に就職内定するも，留年のため棒にふる．以後，紆余曲折を経て，現在フリーランスで執筆/翻訳者として活動中．最近念願のMacユーザーとなる．

梅原　系(うめはら　けい)

1964年11月　兵庫生まれ
血液型はO型，星座は獅子座，干支は辰年と王者の星の下に生まれ，幼少の砌(みぎり)は世界征服を企むが，小学5年の時にマイコンと出会い，進路変更．せっかく入学した某大学計算機学科も，アルバイトに溺れて中退．現在，第一線のプー太郎として，雑誌や単行本の執筆/翻訳/プログラミングなどで日々の糧を得ている．

山田伸一郎(やまだ　しんいちろう)

1966年11月　広島生まれ
関西育ち．電気通信大学大学院博士前期過程に在学中(1990年8月現在)．中学生の時に出会ったMZ80K2により開眼し，Z80の機械語プログラミングに貴重なる青春時代を費やす．その後，ライターとして某パソコン雑誌に連載するが，本領を発揮できず，結局アルバイトとしてアスキーに落ち着き，TOOLS-2からプログラマとして参加する．最近では，OS/2に貴重な研究時間を食われている．なお，本文中の「山田(仮名)」とは別人であると本人は主張している．

■技術協力

水居　浄/上野和明/池田けんしろう/榊　正憲/柳楽直樹

C言語を256倍使うための本

1990年 9 月11日　初版発行
1990年12月21日　第 1 版第 5 刷発行
定価1,100円(本体1,068円)

著　者　福崎俊博／梅原　系／山田伸一郎
発行者　塚本慶一郎
発行所　株式会社アスキー
〒107-24　東京都港区南青山6-11-1スリーエフ南青山ビル
振　替　東京 4 -161144
TEL (03)486-7111 (大代表)
情報TEL (03)498-0299 (ダイヤルイン)
出版営業部TEL (03)486-1977 (ダイヤルイン)

©1990　福崎俊博, 梅原　系, 山田伸一郎

本書は著作権法上の保護を受けています. 本書の一部あるいは全部について (ソフトウェア及びプログラムを含む), 株式会社アスキーから文書による許諾を得ずに, いかなる方法においても無断で複写, 複製することは禁じられています.

制　作　株式会社 ジャパックスインターナショナル
印　刷　共同印刷株式会社

編　集　　中島由弘
本文デザイン　藤田幸代

ISBN 4-7561-0043-0　　Printed in Japan

● 11547

ASCII
256倍

ISBN4-7561-0043-0 C3055 P1100E 定価1,100円(本体1,068円)